TAMPEREEN TEKNILLINEN YLIOPISTO

ANTERO TIIKKAJA

TÖRMÄYSTEN HAVAITSEMINEN KOLMIULOTTEISESSA AVARUUDESSA SELAINYMPÄRISTÖSSÄ

Diplomityö

Tarkastaja: professori Tommi Mikkonen
Tarkastaja ja aihe hyväksytty
Tieto- ja sähkötekniikan tiedekuntaneuvoston kokouksessa 8. kesäkuuta 2011

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
TIIKKAJA, ANTERO: Törmäysten havaitseminen kolmiulotteisessa avaruudessa selainympäristössä
Diplomityö, 56 sivua
Marraskuu 2011
Pääaine: Ohjelmistotuotanto
Tarkastaja: professori Tommi Mikkonen
Avainsanat: JavaScript, törmäysten havaitseminen, rajauslaatikko, WebGL

Webin kehittyminen on mahdollistanut yhä monipuolisemman sisällön esittämisen webissä ja myös kolmiulotteinen grafiikka on siirtynyt yhä enemmän webiin. Tällä hetkellä WebGL mahdollistaa kolmiulotteisen sisällön lisäämisen ja näyttämisen ilman erikseen asennettavia plugin-komponentteja ja tarjoaa siten hyvän lähtökohdan kolmiulotteisen sisällön esittämiseksi webissä. Kehityksen jatkuessa web kilpaillee varteenotettavana sovellusalustana perinteisten sovellusalustojen kanssa.

Törmäysten havaitseminen on monien sovellusalueiden tärkeä osa, sillä ilman törmäystarkastusta useiden sovellusten käyttäminen ei olisi mielekästä. Törmäysten havaitseminen nousee erityisen keskeiseen asemaan interaktiivisuutta vaativissa sovelluksissa, koska törmäysten havaitseminen mahdollistaa vuorovaikutuksen käyttäjän ja virtuaalisen ympäristön välillä.

Työn ongelmana oli, miten selaimet soveltuvat kolmiulotteisen sisällön esittämiseen törmäystarkastelun näkökulmasta ja miten tähän tarkoitukseen luodaan kirjasto JavaScript-kielellä. Kirjastolle asetettiin ohjelmistovaatimuksiksi käytön helppous, siirrettävyys, laajennettavuus ja suorituskyky. Toteutetun kirjaston toimintaa testattiin WebGL:llä toteutetulla testisovelluksella, jota ajettiin eri selaimilla. Testeissä saatujen tulosten perusteella tehtiin päätelmiä luodun kirjaston toiminnasta, käytetyistä selaimista ja web-sovelluskehitysprosessista.

Kirjaston toimintaa testattiin eri selaimilla ja selainten välillä havaittiin selkeitä suorituskykyeroja. Havaintojen perusteella päädyttiin johtopäätöksiin, joiden mukaan web-sovelluskehityksessä sovellusten testaaminen eri selaimilla nousee tärkeään asemaan, koska sovelluksen toiminta voi hidastua huomattavasti selaimesta riippuen ja sovelluksen interaktiivinen ja reaaliaikainen luonne voi kärsiä. Lisäksi tehtiin johtopäätöksiä dokumentoinnin roolista web-sovelluskehityksessä. Web-sovelluskehityksessä käytettävät dynaamiset ohjelmointikielet vaativat hyvän dokumentaation rajapintamäärittelyjen tueksi, jotta toteutettavan kirjaston rajapinnan vaatimat tietotyypit voidaan ilmaista selkeästi kirjaston käyttäjälle ja siten ohjata käyttäjää kirjaston käytössä.

Toteutetun kirjaston arvioinnissa havaittiin, että kirjasto soveltuu hyvin törmäysten havaitsemiseen selainympäristössä, ja kirjaston toteutus vastaa hyvin kirjastolle asetettuja ohjelmistovaatimuksia.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
TIIKKAJA, ANTERO: Collision detection in 3D browser environment
Master of Science Thesis, 56 pages
November 2011
Major: Software Engineering
Examiner: Professor Tommi Mikkonen
Keywords: JavaScript, collision detection, bounding volume, WebGL

The development of the web has made it possible to display more and more diverse content, even 3D graphics, on web pages. Currently WebGL allows 3D graphics to be displayed on a web page without any separate plugins and thus provides a solid base to build 3D content to web. As the development of the web proceeds, web may even compete with the traditional application platforms.

Collision detection is a crucial part of many application areas and without collision detection many applications would be rendered useless. For example the interaction inside or with a virtual environment would not be possible without collision detection.

Research question in this thesis was how the most popular browsers can handle collision detection in 3D environments and how a collision detection library is implemented for browsers using JavaScript. Software requirements for the library were ease of use, portability, extensibility and performance. Implemented library was tested with a test application which utilized WebGL for 3D drawing. The application was run with few of the most popular browsers and, based on the results, conclusions were made of the library, used browsers and application development process for the web environment.

Tests revealed performance differences between used browsers and conclusions were made that testing web applications on different browsers is crucial part of web application development. Sufficient testing of the application can reveal performance issues that would otherwise affect the interactive and realtime nature of the application. Conclusions were made also of the role of documentation when using dynamic programming languages. When developing libraries good documentation is important because it helps the user of the library to use the interfaces as intended.

Final conclusions of the implemented collision detection library were that library serves its purpose well as a collision detection library for browser environment and the library satisfies all of its software requirements.

ALKUSANAT

Kiinnostukseni kolmiulotteiseen grafiikkaan on ohjannut sekä opintojani että vapaa-ajan harrastuksiani jo useita vuosia. Grafiikan lisäksi olen kiinnostunut sen hyödyntämisestä erilaisissa sovelluksissa. Interaktiivisten kolmiulotteisten ympäristöjen ja maailmojen rakentaminen vaatii kuitenkin törmäysten havaitsemista, jotta ympäristön sisäinen ja käyttäjän ja ympäristön välinen vuorovaikutus olisi mahdollista.

Webin kehittymisen myötä kolmiulotteinen grafiikka on myös siirtynyt yhä enemmän webiin ja siten törmäysten havaitseminen noussee tärkeäksi osaksi tulevaisuuden web-sovelluksia. Olen tehnyt kandidaatintyöni törmäysten havaitsemiseen liittyen ja koin, että työn laajentaminen diplomityöksi ja selainympäristöjä koskevaksi oli hyvä valinta. Tekemäni diplomityö on saanut minut yhä kiinnostuneemmaksi kolmiulotteisista ympäristöistä ja työ onkin vienyt minut mukanaan.

Haluan kiittää työni ohjaajaa Tommi Mikkosta hyvästä ohjauksesta ja palautteesta, joka on auttanut minua työni tekemisessä ja parantamisessa.

Tampereella 24.10.2011

Antero Tiikkaja

SISÄLLYS

1 JOHDANTO

Webin kehittyminen on mahdollistanut yhä monipuolisemman sisällön esittämisen webissä. Kehityksen jatkuessa web noussee yhdeksi varteenotettavaksi sovellusalustaksi, joka kilpailee perinteisten sovellusalustojen ja käyttöjärjestelmien kanssa. Jo nyt web tukee interaktiivisia, reaaliaikaisia ja usean käyttäjän välisiä sovelluksia. Websovelluskehityksessä käytettävät dynaamiset ohjelmointikielet eivät kuitenkaan vastaa ominaisuuksiltaan täysin tavanomaiseen sovelluskehitykseen käytettäviä staattisia kieliä. Tästä johtuen sovellusten ja kirjastojen kehittäminen webiin poikkeaa perinteisten sovellusten ja kirjastojen kehittämisestä.

Sovellusten monipuolisuuden lisäksi myös webin graafiset ominaisuudet ovat kehittyneet. Alkuaikoinaan web tuki kaksiulotteisia kuvia ja animaatiota, mutta kehityksen myötä myös kolmiulotteinen grafiikka on siirtynyt webiin. Aluksi kolmiulotteista sisältöä saatiin lisättyä erillisten plugin-komponenttien avulla, mutta kehityksen suuntana on pyrkiä luopumaan näistä erikseen asennettavista komponenteista. Tällä hetkellä WebGL mahdollistaa kolmiulotteisen sisällön lisäämisen ja näyttämisen ilman erikseen asennettavia plugin-komponentteja.

Törmäysten havaitseminen on monien sovellusalueiden tärkeä osa, sillä ilman törmäystarkastusta useiden sovellusten käyttäminen ei olisi mielekästä. Tällaisia sovellusalueita ovat esimerkiksi fysikaalinen mallinnus, animaatio, robotiikka ja pelit. Viihde- ja hyötysovellusten yhtenä lähtökohtana on usein reaalimaailman mallintaminen mahdollisimman uskottavasti, ja niissä törmäysten havaitsemista tarvitaan ympäristön sekä sen sisältämien kappaleiden välisen vuorovaikutuksen todentuntuiseen mallintamiseen. Vaikka varsinainen törmäykseen reagointi ei sisälly törmäystarkastukseen, on törmäyksen havaitseminen kuitenkin lähtökohta koko törmäyksen mallintamiselle.

Törmäysten havaitseminen nousee erityisen keskeiseen asemaan muun muassa peleissä ja virtuaalitodellisuudessa, mutta myös muissa interaktiivisuutta vaativissa sovelluksissa. Törmäysten havaitseminen on mahdollistanut esimerkiksi käyttäjän ja ympäristön välisen vuorovaikutuksen. Ilman vuorovaikutusta käyttäjän olisi mahdotonta toimia pelimaailmassa tai virtuaaliympäristössä, ja näin ollen törmäysten havaitseminen onkin mahdollistanut näiden sovellusten mielekkään toiminnan.

Tässä diplomityössä tutkittiin törmäysten tarkastelua kolmiulotteisessa selainympäristössä. Diplomityön teknisenä kontribuutiona toteutettiin JavaScript -kirjasto rajauslaatikoiden sisään rajattujen kappaleiden välisten törmäysten havaitsemiseksi. Tarkoituksena oli selvittää, miten selaimet soveltuvat kolmiulotteisen sisällön esittämiseen törmäystarkastelun näkökulmasta ja miten tähän tarkoitukseen luodaan kirjasto JavaSc-

ript-kielellä. Luodun kirjaston toimintaa testattiin WebGL:llä toteutetulla testisovelluksella, jota ajettiin eri selaimilla.

Luvussa 2 käydään läpi lyhyesti webin kehitys sovellusalustana ja kolmiulotteisen sisällön esittäminen webissä WebGL-tekniikkaa käyttämällä. Luvussa käsitellään myös JavaScript-kielen perusteita. Luvussa 3 käydään läpi kolmiulotteiseen grafiikkaan sekä törmäystarkasteluun liittyviä perusteita. Luvussa 4 käydään läpi toteutetun törmäystarkastelukirjaston arkkitehtuuri, kirjaston käyttäminen ja siihen liittyvät rajoitukset. Luvussa 5 esitellään toteutettu esimerkkiohjelma ja arvioidaan saatuja tuloksia. Lopuksi luvussa 6 esitetään yhteenveto diplomityöstä.

2 KOLMIULOTTEINEN SISÄLTÖ WEBISSÄ

Webin kehittyminen yhdessä laskentatehon kasvun ja graafisten ominaisuuksien kehittymisen kanssa ovat mahdollistaneet yhä monipuolisemman sisällön näyttämisen webissä, joka on muuttumassa pelkästä tiedon esittämisestä yhä enemmän kokonaisvaltaiseksi sovellusalustaksi (Anttonen et al. 2011). Web on tukenut interaktiivisen sisällön esittämistä jo vuodesta 1995, jolloin JavaScript-kieli esiteltiin. Tästä huolimatta webin sisältö on ollut pääasiassa kaksiulotteista ja se on pohjautunut selainten tukemaan plugin-tekniikkaan.

Tässä luvussa käydään lyhyesti läpi webin kehitys sisällön näkökulmasta, JavaScript-kielen perusteet sekä siihen liittyviä erityispiirteitä ja kolmiulotteisen sisällön esittäminen WebGL-tekniikkaa käyttämällä.

2.1 Webin kehitys sovellusalustana

Webin kehityksessä voidaan tunnistaa selvästi kolme eri vaihetta. Aluksi webin sisältö muodostui tekstistä ja kuvista ja sivuilla siirtyminen tapahtui hyperlinkkien avulla. Sisältö oli siis staattista ja käyttäjän vuorovaikutus rajattua. Seuraavassa vaiheessa käyttäjän vuorovaikutusmahdollisuuksia lisättiin plugin-komponenttien avulla. Tunnetuimpia plugin-komponentteja ovat muun muassa Applen QuickTime sekä Adoben Flash. Näiden avulla oli mahdollista esittää hyvin monipuolista sisältöä, kuten grafiikkaa, ääntä ja videota. Vuonna 1995 julkaistiin JavaScript-skriptikieli, jonka avulla voitiin helposti tuottaa animoitua ja interaktiivista sisältöä. Kolmannessa vaiheessa webin kehitys on keskittynyt yhä enemmän yhteisöllisyyden ja vuorovaikutuksen lisäämiseen. (Taivalsaari et al. 2008.)

Webissä painotus on yhä enemmän interaktiivisen sisällön tarjoamisessa, mistä syystä törmäysten tarkastelu nousee tärkeään osaan sisältöä tuotettaessa. Tekniikan kehittyminen mahdollistaa yhä monimutkaisemman sisällön esittämisen websivuilla. Alkuaikojen kaksiulotteiset kuvat ovat saaneet rinnalleen kolmiulotteista ja reaaliaikaista grafiikkaa, mikä asettaa vaatimuksia sovellusten toteuttamiselle.

Törmäysten tarkastelun näkökulmasta uuden ulottuvuuden lisääminen monimutkaistaa laskutoimituksia tapauksesta riippuen. Törmäysten havaitseminen tulisi tapahtua silti nopeasti, jotta sovellus ei menetä interaktiivista luonnettaan (Cohen et al. 1995). Tästä syystä on tärkeää, että selaimet soveltuvat hyvin tämän sisällön esittämiseen myös törmäystarkastelun näkökulmasta. Selaimessa tehtävän törmäystarkastelun tulee siis toimia nopeasti, mutta myös tarkasti, jotta koettu käyttökokemus olisi mahdollisimman hyvä.

Web sisältää yhä heikkouksia, jotka jarruttavat webin kehitystä sovellusalustana. Taivalsaari et al. (2008) mukaan tunnistettuja heikkouksia ovat muun muassa weboh-

jelmoinnissa käytettävät dynaamiset ohjelmointikielet, sovellusten suorittamiseen käytettävien virtuaalikoneiden suorituskyky ja selainten väliset erot. Vaikkakin nämä heikkoudet on listattu koskemaan webin kehitystä sovellusalustana, on ne kuitenkin huomioitava myös uuden sisällön kehittämisessä.

Dynaamiset ohjelmointikielet vaativat erilaisen lähestymistavan ohjelmistokehitykseen verrattuna staattisiin ohjelmointikieliin (Taivalsaari et al. 2008). Staattisten ohjelmointikielten, kuten C++:n ja Javan, avulla tuotetaan sovelluksia, joiden koodi ja looginen rakenne pysyvät muuttumattomina ohjelman suorituksen aikana. Ainoa keino niiden muuttamiseen on tehdä muutoksia itse koodiin ja kääntää ohjelma uudelleen. Dynaamista ohjelmointikieltä käytettäessä ohjelman käyttäytymistä voidaan muuttaa ajon aikana luomalla esimerkiksi moduulien nimiä, luokkia ja funktioita kesken ohjelman suorituksen. (Paulson 2007.)

JavaScript on kaikkien selainten tukema kieli ja sen ajaminen selaimessa vaatii virtuaalikoneen käyttämistä. JavaScriptillä tehtävässä sovelluskehityksessä on siis huomioitava, että JavaScriptin ajamiseen vaadittava virtuaalikone on huomattavasti hitaampi verrattuna puhtaasti käyttäjän koneelta suorittamiseen (Taivalsaari et al. 2008).

Selainten väliset erot voivat aiheuttaa sovelluksen toimimisen erilailla eri selaimissa. Standardoinnilla on pyritty vähentämään yhteensopivuusongelmia, mutta standardien käyttöä ei vaadita. Tästä johtuen osa selaintoimittajista ei noudata standardoituja toimintatapoja, vaan kannattavat omia ratkaisujaan. (Taivalsaari et al. 2008.)

2.2 JavaScript

Netscape Communications Corporation julkaisi JavaScript-kielen alun perin vuonna 1995 ja viimeisin kielen määrittely on vuodelta 1999. JavaScript on edelleen laajasti käytössä ja sitä tukevat kaikki webselaimet (Crockford 2008, s. 4). Yksi JavaScriptin tärkeimmistä ominaisuuksista on mahdollistaa interaktiivisen toiminnallisuuden lisääminen websivuille käyttämällä selainten API:a, DOM-puuta (W3C 2005). JavaScript on heikosti tyypitetty, objektipohjainen ja ajon aikana evaluoitu, tulkattava kieli.

2.2.1 Heikko tyypitys

Heikko tyypitys tarkoittaa, että käytetyllä muuttujalla ei ole kiinteästi määriteltyä datatyyppiä. Tästä seuraa, että muuttujaan voidaan sijoittaa vapaasti kaiken tyyppistä tietoa. Vaikka muuttujalle ei suoraan määritetä tyyppiä, voidaan muuttujan tyyppi kysyä erillisellä `typeof`-operaattorilla. Taulukossa 2.1. on esitetty JavaScriptin käyttämät perustyypit ja taulukossa 2.2. on esitetty `typeof`-operaattorin palauttamat paluuarvot.

Heikko tyypitys on osaltaan syynä siihen, että JavaScript-ohjelmien laadun valvonta on hankalaa. Tästä syystä JavaScript-ohjelmien toteutuksessa tulisi noudattaa järjestelmällisyyttä, jotta heikosta tyypityksestä johtuvat negatiiviset vaikutuksen ohjelman laatuun saadaan poistettua jo toteutusvaiheessa. (Crockford 2008, s. 94.)

Taulukko 2.1. *JavaScriptin käyttämät perustyypit (muokattu lähteestä Crockford 2008, s. 20).*

Varattu sana	Tyyppi	Esimerkki
number	numero	var numero = 4;
string	merkkijono	var mjono = ”Merkkijono”
boolean	totuusarvo	var totta = true;
null	tyhjä	var tyhja = null;
undefined	määrittelemätön	var maarittelematon;
object	objekti	var objekti = {};

Taulukko 2.2. *Typeof-operaattorin palauttamat paluuarvot (muokattu lähteestä Crockford 2008, s. 16).*

Varattu sana	Tyyppi	Esimerkki	
number	numero	typeof numero;	// = number
string	merkkijono	typeof mjono;	// = string
boolean	totuusarvo	typeof totta;	// = boolean
undefined	määrittelemätön	typeof maarittelematon;	// = undefined
object	objekti	typeof objekti;	// = object
function	funktio	typeof function(){}	// = number

On huomattava, että jos muuttujan tyyppi on `null`, palauttaa `typeof`-operaattori paluuarvon `object`. Myös jos muuttujaan on sijoitettu lista, `Array`, on paluuarvo `object`. (Crockford 2008, s. 16.)

Muuttuja määritellään käyttämällä varattua sanaa `var`, jonka jälkeen annetaan muuttujan nimi ja mahdollinen alustusarvo. Heikosta tyypityksestä johtuen muuttujia voidaan sijoittaa toisiinsa riippumatta siitä, minkä tyyppistä dataa muuttujissa on ennen sijoittamista. Listauksessa 2.1. on esitetty laillinen JavaScript ohjelma, joka antaa esimerkin muuttujien käyttämisestä ja heikon tyypityksen tuomista mahdollisuuksista.

Listaus 2.1. *Laillinen JavaScript ohjelma.*

```
var numero = 4.0;
var merkkijono = "Hei Maailma!";
numero = merkkijono;

function tulosta(parametri) {
    alert(parametri);
}

tulosta(numero); // Tulostaa "Hei Maailma!"
numero = 11;
--numero;
tulosta(numero); // Tulostaa 10
```

Kuten listauksesta 2.1. nähdään, myös funktioiden parametrit ovat tyypittömiä. Tämä mahdollistaa saman funktion käyttämisen monentyyppiselle tiedolle ilman funktion

määrittelyä jokaiselle tyypille erikseen. Esimerkiksi C++:ssa vastaava toiminnallisuus vaatii funktioiden ylikuormittamista.

Heikko tyypitys mahdollistaa helposti eri ratkaisutapojen kokeilun, mutta se tekee ohjelmien testaamisesta vaikeampaa tilanteissa, joissa muuttujan tyypillä on väliä. Ennen muuttujan käyttöä täytyy siis tarvittaessa varmistua siitä, että muuttuja sisältää oikean tyyppistä tietoa. Sama pätee myös funktioiden parametreihin. Ongelma korostuu tilanteissa, joissa käytössä on esimerkiksi kolmannen osapuolen tuottama JavaScript-kirjasto. Jos kirjaston rajapinta on huonosti dokumentoitu, ei kirjaston käyttäjällä ole välttämättä mitään keinoa varmistua siitä, minkä tyyppistä tietoa rajapintafunktiot vaativat.

Sopimussuunnittelussa (Rintala & Jokinen 2005, s. 197–198) jaetaan toiminnan vastuuta kutsujan ja toteutuksen kesken. Rajapinnan toteutus lupaa toimia jokaisen rajapinnan palvelun osalta tietyllä tavalla, kun rajapintaa käytetään määritellyllä tavalla. Rajapinnan määrittelyssä on siis mukana tieto rajapinnan sallituista käyttötavoista. Tähän määriteltävään tietoon kuuluvat esimerkiksi funktioiden parametrit, kutsujärjestykset ja tietotyyppien arvorajat. Rajapintaa käyttävä ohjelmoija lupaa puolestaan käyttää rajapinnan palveluita ainoastaan niiden määrittelyn mukaisesti.

Rajapinnan yksittäisen palvelun tasolla sopimus tehdään määrittelemällä palvelulle esi- ja jälkiehto. Molemmat ovat matemaattisesti määriteltyjä, predikaattilogiikan loogisia lausekkeita, joiden noudattamiseen palvelu sitoutuu. Palvelu lupaa, että jos palvelun suorituksen alkaessa esiehto on voimassa, niin vastaavasti palvelun päättyessä myös jälkiehto on voimassa. Jos taas palvelua kutsuttaessa esiehto ei toteudu, niin palvelu toimii määrittelemättömästi, jolloin jälkiehdonkaan ei tarvitse toteutua palvelusta palattaessa. (Rintala & Jokinen 2005, s. 198.)

Sopimussuunnittelu ei kuitenkaan skaalaudu ohjelmiston ylimmän tason moduulien rajapintoihin, koska esi- ja jälkiehtojen tarkkaan määrittelyyn tarvitaan usein paljon informaatiota. Tästä syystä tavallisimmissa ohjelmistoprojekteissa sopimusten matemaattista määrittelyä ei yleensä käytetä. Sopimussuunnittelun periaatteiden tunteminen ja osittainen noudattaminen on kuitenkin pelkkää sanallista rajapintakuvausta eksaktimpi. Suurimpana hyötynä sopimussuunnittelu, edes osittain noudatettuna, auttaa ennen kaikkea rajapintojen suunnittelussa, koska se pakottaa ja ohjaa rajapintojen huolelliseen dokumentointiin. Lisäksi rajapinnan toteutukseen vaadittava kokonaiskoodimäärä on yleensä pienempi, koska toteutuksessa ei tarvitse varautua määrittelyn ulkopuolisiin tilanteisiin. (Rintala & Jokinen 2005, s. 198–200.)

JavaScript-kirjastoja toteutettaessa vähintään sopimussuunnittelun osittainen noudattaminen on hyvin tärkeässä asemassa. Kirjaston luojan tulee selkeästi määritellä rajapintafunktioiden vaatiman tiedon tyyppi ja funktioiden paluuarvojen muoto. Kirjaston käyttäjän tulee puolestaan vastata siitä, että hän käyttää kirjastoa sen määrittelyn mukaisesti.

2.2.2 Objektipohjaisuus

JavaScriptissä listat, funktiot, säännölliset lausekkeet ja objektit ovat objekteja. Objekti on ominaisuussäiliö, jossa ominaisuudella on nimi ja arvo. Ominaisuuden nimi on tyypiltään merkkijono, mutta sen pituudelle ei ole vaatimuksia. Tästä johtuen ominaisuuden nimi voi olla myös tyhjä merkkijono. Ominaisuuden arvo voi olla mikä tahansa JavaScript-arvo paitsi `undefined`. Objektit soveltuvat hyvin datan keräämiseen ja organisointiin, mutta myös puumaisten rakenteiden, kuten graafien esittämiseen. (Crockford 2008, s. 20.)

Objekti voidaan luoda esimerkiksi käyttämällä `new`-operaattoria, objektiliteraalia tai kutsumalla mitä tahansa funktiota, joka palauttaa objektin (Crockford 2008, s. 52) Objektiliteraali on aaltosulkeiden sisään sijoitettu, mahdollisesti myös tyhjä, lista objektin ominaisuuksista. Luomalla objekti käyttämällä objektiliteraalia, voidaan objektille täten määritellä helposti myös objektin sisältö sen luomisvaiheessa. (Crockford 2008, s. 20.)

Listauksessa 2.2. on esitetty objektien luominen `new`-operaattorilla ja objektiliteraalin avulla.

Listaus 2.2. *Objektien luominen sekä `new`-operaattorilla että objektiliteraalilla.*

```
var objekti_1 = new Object();
objekti_1.nimi = "Jukka";
objekti_1.ika = 20;

var objekti_2 = {
    nimi: "Pekka",
    ika: 24
}

var objekti_3 = {};
objekti_3.nimi = "Kalle";
objekti_3.ika = 30;
```

Objektit kannattaa luoda objektiliteraalin avulla, koska `new`-operaattorin käyttö on virhealttiimpaa. Jos objekti on tarkoitus luoda `new`-operaattorilla, mutta `new`-sana unohtuu ennen rakentajafunktiota, on tuloksena tavallinen funktiokutsu. Tässä tilanteessa funktio kirjoittaa yli globaaleja muuttujia, kun se yrittää alustaa sille kuuluvia paikallisia muuttujia. (Crockford 2008, s. 114.)

Objektin arvojen lukemiseen voidaan käyttää sekä kulmasulkeita että pistenotaatiota (esimerkki listauksessa 2.3.). Yritys lukea arvoa, jota objektilla ei ole, muodostaa arvon `undefined`. (Crockford 2008, s. 21).

Listaus 2.3. *Objektin arvojen lukeminen kulmasulkeita ja pistenotaatiota käyttämällä.*

```
var nimi = objekti_1["nimi"];  // nimi = "Jukka"
var ika = objekti_2.ika;       // ika = 24
var osoite = objekti_3.osoite; // osoite = undefined
```

Crockford (2008, s. 21) suosittelee pistenotaation käyttämistä, koska sen lukeminen on helpompaa ja se vie kirjoitettuna vähemmän tilaa. Pistenotaatiota voidaan käyttää silloin, kun haettavan arvon nimi on vakio ja nimi ei ole varattu JavaScript-sana.

2.2.3 Ajonaikainen evaluointi

JavaScript on tulkattava kieli, ja JavaScript-ohjelma evaluoidaan vasta ajon aikana. Ero käännettäviin, vahvasti tyypitettyihin kieliin, kuten C++:aan, on se, että ohjelma tarkastetaan kokonaisuudessaan vasta silloin kun se ajetaan. Erillisen käännösvaiheen puuttuminen ja heikon tyypitykseen mahdollistama datatyyppien vapaa käyttö voi tehdä JavaScript-ohjelmien testaamisesta hitaampaa ja vaikeampaa (Crockford 2008, s. 3).

Vahvasti tyypitetyissä, käännettävissä kielissä kääntäjä havaitsee esimerkiksi muuttujien tyyppivirheet, jolloin ne voidaan korjata ennen ohjelman ajamista (Crockford 2008, s. 3). JavaScriptin ajon aikainen evaluointi tutkii muuttujien datatyyppejä ohjelman suorituksen aikana, mikä aiheuttaa tehohäviötä ohjelman suorituksessa. Toisaalta tarvittavan ohjelmakoodin määrä pienenee ja koodia kirjoittaessa ei tarvitse huomioida esimerkiksi tyyppimuunnoksia. Tämä siirtää ohjelmointityötä pienten yksityiskohtien huomioimisesta enemmän nopean, tuottavan ja luovan prosessin suuntaan. (Paulson 2007.)

2.2.4 Olio-ohjelmointi JavaScriptillä

JavaScript tarjoaa keinoja toteuttaa olio-ohjelmointia, vaikka kieli itsessään ei perustu olio-ohjelmointiin. Jatkossa käytetään olio-ohjelmoinnissa yleisesti käytettyjä termejä, kuten luokka ja olio, vaikka JavaScriptissä niitä ei periaatteessa ole. Yleisten termien käyttö on kuitenkin perusteltua selkeyden parantamiseksi ja koska JavaScriptin avulla voidaan tuottaa olio-ohjelmointia vastaavaa toiminnallisuutta.

Crockford (2008, s. 52–55) esittää yhden tavan toteuttaa olio-ohjelmointia JavaScriptillä. Oliot luodaan funktioina, jotka käyttävät funktioiden sulkeuma-ominaisuutta hyväkseen piilottamaan olioiden tilan. Sulkeuma tarkoittaa, että funktiot pääsevät käsiksi niiden funktioiden parametreihin ja muuttujiin, joiden sisällä ne ovat määritelty. Näin voidaan luoda olioita, joiden tilaan päästään käsiksi vain erikseen määritellyn rajapinnan avulla. Myös olion tilaa voi muuttaa ainoastaan tätä rajapintaa. Listauksessa 2.4. on esitetty yksinkertaisen nisäkäs-olion määrittely, kun käytetään funktioiden sulkeuma-ominaisuutta piilottamaan olion tila.

Listaus 2.4. *Nisäkäs-olion määrittely.*

```
var nisakas = function(sanoma) {
    var olio = {};
    var mSanoma = sanoma || "Minä olen nisäkäs!";

    var aantele = function() {
        alert(mSanoma);
    }
```

```
    olio.aantele = aantele;

    return olio;
```

Nisäkäs luodaan siis funktiona, jonka sisällä määritellään olion jäsenmuuttujia vastaavat muuttujat, tässä tapauksessa siis `mSanoma`, joka pitää sisällään merkkijonon. Funktion sisällä määritellään myös rajapinta, jonka kautta päästään käsiksi olion tilaan. Tässä tapauksessa oliota voidaan pyytää tulostamaan sanomansa ruudulle kutsumalla oliolle `aantele`-metodia. Määrittelemällä `aantele` ensin tavalliseksi muuttujaksi ja sijoittamalla se erikseen palautettavat olion metodiksi voidaan parantaa turvallisuutta (Crockford 2008, s. 53).

Nisäkäs on toteutettu tehdas-tyyppisesti, jolloin nisäkkään luominen tapahtuu kutsumalla `nisakas`-funktiota ilman `new` operaattoria. Luotua nisäkästä voidaan pyytää ääntelemään kutsumalla nisäkkäälle sen `aantele`-funktiota. Esimerkki kutsujen toteuttamisesta on esitetty listauksessa 2.5.

Listaus 2.5. *Nisäkäs-olioiden luominen ja käyttäminen.*

```
var nisakas_1 = nisakas();
var nisakas_2 = nisakas(”Hei Maailma!”);

nisakas_1.aantele(); // Tulostaa ”Minä olen nisäkäs!”
nisakas_2.aantele(); // Tulostaa ”Hei Maailma!”
```

Käytetty tehdasmallinen olioiden luominen tukee myös olioiden periytymistä. Listauksessa 2.6. on esitetty miten koiraolio määritellään käyttäen apuna nisäkäsoliota.

Listaus 2.6. *Koira-olion periyttäminen nisäkäs oliosta.*

```
var koira = function(sanoma, nimi) {
    var olio = nisakas(sanoma);
    var mNimi = nimi || ”Rekku”;

    var sano_nimi = function() {
        alert(”Nimeni on ” + mNimi);
    }

    olio.sano_nimi = sano_nimi;

    return olio;
}
```

Rakentajafunktion käyttämät parametrit voidaan esittää yhden rakentajaobjektin avulla käyttämällä alakohdassa 2.2.2. mainittua objektipohjaisuutta. Rakentajaobjektin käyttäminen on kätevää tilanteissa, joissa käytettävä rakentajafunktio tarvitsee suuren määrän parametreja. Käyttämällä rakentajaobjektia, jokaista parametria ei tarvitse listata

funktion parametrilistaan erikseen, vaan rakentajaobjektista poimitaan tarvittava tieto oliota luotaessa. Listauksessa 2.7. on esitetty miten koiraolion rakentajalle menevät parametrit voidaan sisällyttää rakentajaobjektiksi.

Listaus 2.7. *Koira-olion rakentajaa varten luotu rakentajaobjekti.*

```
var rakentajaobjekti = {};
rakentajaobjekti.sanoma = ”Hei Maailma!”;
rakentajaobjekti.nimi = ”Musti”;
var koira_olio = koira(rakentajaobjekti);
```

Listauksessa 2.8. on esitetty koiraolion rakentaja käyttämällä rakentajaobjektia parametrilistan sijaan.

Listaus 2.8. *Koira-olion rakentaja käyttämällä rakentajaobjektia.*

```
var koira = function(rakentajaobjekti) {
    var olio = nisakas(rakentajaobjekti.sanoma);
    var mNimi = rakentajaobjekti.nimi || ”Rekku”;

    var sano_nimi = function() {
        alert(”Nimeni on ” + mNimi);
    }

    olio.sano_nimi = sano_nimi;

    return olio;
}
```

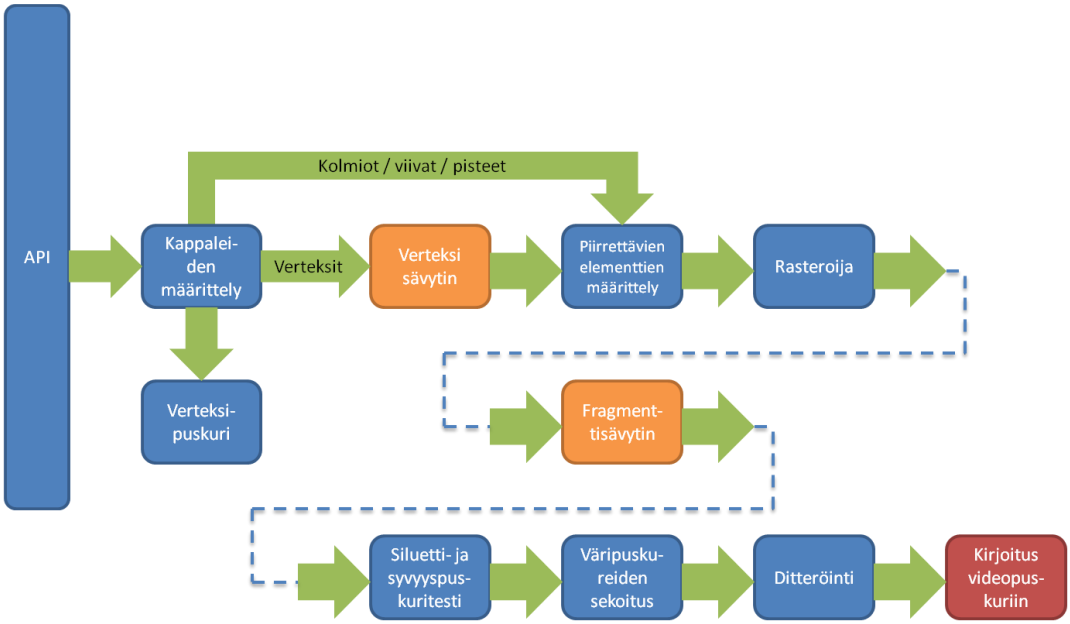
Periytymiseen liittyen on huomattava, että JavaScriptissä periytyminen ei perustu luokkiin vaan prototyyppeihin. Tämä tarkoittaa, että JavaScriptissä luotava olio ei ole ilmentymä luokasta, joka perii isäntäluokan, vaan objektit perivät suoraan toisia objekteja. Heikko tyypitys saa aikaan JavaScript-objektien tapauksessa sen, että objektia käytettäessä ei ole väliä mistä objekteista kohde objekti periytyy, vaan mitä objekti voi käytännössä tehdä. (Crockford 2008, s. 46.)

2.3 Kolmiulotteisen sisällön esittäminen WebGL-tekniikkaa käyttäen

Selaimiin asennettavat Plugin-komponentit mahdollistivat vuorovaikutuksen kasvun lisäksi myös kolmiulotteisen sisällön esittämisen. Esimerkiksi Flash (Adobe), O3D (O3D), VRML (VRML97) ja X3D (X3D) ovat mahdollistaneet komiulotteisen sisällön esittämisen selaimeen asennettavan pluginin avulla (Anttonen & Salminen 2011).

Kolmiulotteisen sisällön kehityksen seuraava askel on WebGL, joka mahdollistaa kolmiulotteisen sisällön näyttämisen selaimessa ilman erillisiä plugineja. Erikseen asennettavan pluginin sijaan WebGL käyttää piirtopintanaan HTML 5-standardiin kuuluvaa Canvas-elementtiä. Selaimen Document Object Model -rajapinta (DOM) mahdollistaa

pääsyn WebGL:n tarvitsemiin resursseihin ja JavaScriptillä toteutettu API mahdollistaa OpenGL-kutsujen tekemisen ja siten WebGL:n käytön. WebGL perustuu Khronos groupin OpenGL ES 2.0 -spesifikaatioon ja se käyttää GLSL-sävytinkieltä tarjoten siten ohjelmoitavan liukuhihnan (esitetty kuvassa 2.1.). (Khronos Group 2011.)



Kuva 2.1. *OpenGL ES 2.0 liukuhihna (muokattu lähteestä Khronos Group).*

Seuraavaksi käydään läpi yksinkertaisen WebGL-sovelluksen toteuttamiseen vaadittavat toimenpiteet. Sovelluksen rakenne on jaettu kahteen osaan, joista ensimmäisessä WebGL asetetaan käyttökuntoon ja toisessa käydään läpi varsinaisen piirron toteuttaminen.

2.3.1 Alustus

Ennen kuin WebGL:n avulla voidaan piirtää, on sivulle lisättävä `Canvas`-elementti, määriteltävä websivun `<body>`-osion `onload`-attribuutille arvo ja määriteltävä käytettävät sävyttimet.

`Canvas`-elementti toimii WebGL:n piirtopintana ja lisätään websivun `<body>`-osioon. WebGL:n käyttäminen itsessään ei vaadi `<body>`-osion sisällöksi muuta (Anttonen & Salminen 2011). `Canvas`-elementille on annettava `id`-tunnus, jonka avulla elementtiin voidaan viitata sovelluksen sisältä, sekä lisäksi piirtoalueen mitat. Esimerkki `Canvas`-elementin asettamisesta sivulle on nähtävissä Listauksessa 2.9.

Listaus 2.9. *Canvas-elementin lisääminen websivulle WebGL piirtoa varten (muokattu lähteestä Anttonen & Salminen 2011).*

```
<body onload=”initGL();”>
<canvas id=”glCanvas” style=”border: none;” width=”512”
```

```
height="512"></canvas>
</body>
```

Listauksessa 2.9. on myös esitetty `<body>`-osion `onload`-attribuutin asettaminen. Attribuutin avulla voidaan asettaa websivun latautumisen yhteyteen toiminnallisuutta. Kuvassa asetettu arvo tarkoittaa, että kun websivun `<body>`-osio on ladattu, kutsutaan `initGL()`-nimistä JavaScript-funktiota (Anttonen & Salminen 2011). Funktion määrittely tehdään `<head>`-osiossa ja se sijoitetaan `<script>`-elementin sisään. Tätä samaa `<script>`-elementtiä voidaan käyttää myös muiden sovelluksen toimintaan liittyvien funktioiden määrittelyyn. Listauksessa 2.10. on esitetty `<head>`-osion runko, joka sisältää `initGL()`-funktion määrittelyn ja funktiot sovelluksen muuhun toimintaan.

Listaus 2.10. *<head> osion runko ja initGL()-funktion määrittely (muokattu lähteestä Anttonen & Salminen 2011).*

```
<head><script type="text/javascript">
var gl;
function initGL() {
    var canvas = document.getElementById("glCanvas");
    try {
        gl = canvas.getContext("experimental-webgl");
        gl.viewport(0, 0, canvas.width, canvas.height);
        initShaders();
        initBuffers();
    }
    catch (e) {
        alert("Could not initialize WebGL!");
    }
    drawScene();
}
function initShaders() {
...
}
function initBuffers() {
...
}
function drawScene() {
...
}
</script></head>
```

Listauksessa 2.10. näkyvän `initShaders()`-funktion toteutus lukee, kääntää ja linkittää käytetyt sävyttimet. `InitBuffers()`-funktio luo piirrettävät kappaleet ja `drawScene()`-funktio piirtää kappaleista luodun ympäristön ruudulle.

Ennen piirtämisen aloittamista, täytyy määritellä käytettävät verteksi- ja fragmenttisävyttimet. Verteksisävytin korvaa kiinteät monikulmioiden kulmapisteiden muunnok-

set ja kulmapistekohtaisen valaistuksen laskennan, ja sen tuottamat tulokset interpoloidaan yksittäisille kuvapisteille ja ne toimivat syötteenä fragmenttisävyttimelle (Puhakka 2008, s. 386). Sävyttimien määrittely tehdään websivun `<head>`-osiossa erillisissä JavaScript-elementeissä. Listauksessa 2.11. on esitetty esimerkit sekä verteksi- että fragmenttisävyttimestä.

Listaus 2.11. *Esimerkit verteksisävyttimestä (yllä) ja fragmenttisävyttimestä (muokattu lähteestä Anttonen & Salminen 2011).*

```
<script id=”shader-vs” type=”x-shader/x-vertex”>
attribute vec3 aPosition;
uniform mat4 mMMatrix;
uniform mat4 mPMatrix;

void main(void) {
    gl_Position = mPMatrix*mMMatrix*vec4(aPosition, 1.0);
}
</script>

<script id=”shader-fs” type=”x-shader/x-fragment”>
#ifdef GL_ES
precision highp float;
#endif
void main(void) {
    gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);
}
</script>
```

Esimerkkisävyttimet suorittavat syötemonikulmion kulmapisteille tarvittavat geometriset muunnokset ja piirtävät monikulmion puhtaan valkoisella värillä. Sävyttimet luetaan, käännetään ja linkitetään ennen niiden käyttämistä. Listauksen 2.10. tapauksessa nämä toimenpiteet suoritetaan `initShaders()`-funktiossa. Esitettyjen sävytinten toiminta sekä niiden sitominen osaksi WebGL-sovellusta on esitetty laajemmin lähteessä Anttonen & Salminen (2011).

2.3.2 WebGL:n käyttäminen

Kun WebGL on asetettu käyttökuntoon, määritellään piirrettävät mallit. Määrittelyssä mallien geometria tallennetaan puskureihin (engl. buffers), joita käyttämällä WebGL piirtää ne ruudulle. Puskurit voidaan määritellä esimerkiksi `initBuffers()`-nimisessä funktiossa. Mallille tulee määritellä vähintään verteksipuskuri, joka sisältää kappaleen monikulmion kulmapisteiden koordinaatit kolmiulotteisessa avaruudessa. Puskureihin voidaan tallentaa kulmapisteiden lisäksi esimerkiksi kulmapistekohtaisia värejä, tekstuurikoordinaatteja tai normaalivektoreita. Jokaista lisäinformaatiota varten on kuitenkin luotava oma puskurinsa. Lisäksi käytetyt sävyttimet on muokattava tukemaan niitä. (Anttonen & Salminen 2011.)

Listauksessa 2.12. on esitetty verteksipuskurin määrittely neliölle `initBuffers()`-funktion sisällä.

Listaus 2.12. *Verteksipuskurin määrittely neliölle (muokattu lähteestä Anttonen & Salminen 2011).*

```
var squarePosBuffer;
function initBuffers() {
    squarePosBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, squarePosBuffer);
    var vertices = [1.0, 1.0, 0.0,
                    -1.0, 1.0, 0.0,
                     1.0, -1.0, 0.0,
                    -1.0, -1.0, 0.0];
    gl.bufferData(gl.ARRAY_BUFFER,
                  new Float32Array(vertices),
                  gl.STATIC_DRAW);
    squarePosBuffer.itemSize = 3;
    squarePosBuffer.numberOfItems = 4;
}
```

Puskuri on ensin luotava kutsumalla `gl.createBuffer()`-funktiota, jonka jälkeen luotu puskuri kiinnitetään WebGL-ympäristöön `gl.bindBuffer()`-funktiolla. Seuraavaksi määritellään mallin geometria kolmiulotteisessa avaruudessa luettelemalla mallin kulmapisteiden koordinaatit ja sitomalla ne luotuun puskuriin `gl.bufferData()`-kutsulla. Lopuksi puskurille asetetaan kaksi piirrossa tarvittavaa parametria, `itemSize` ja `numberOfItems`. `ItemSize`-parametri kertoo yksittäisen kulmapisteen pituuden. Listauksessa 2.12. tapauksessa `itemSize` asetetaan arvoon 3, koska jokainen kulmapiste sisältää x-, y- ja z-koordinaatin. `NumberOfItems`-parametriin tallennetaan kulmapisteiden lukumäärä. Neliön tapauksessa tämä on siis 4. (Anttonen & Salminen 2011.)

Kappaleiden määrittelyn jälkeen voidaan luotuja puskureita käyttää geometrian piirtämiseen. Jotta piirto voidaan toteuttaa, täytyy määritellä vielä käytettävä projisointitapa sekä mahdolliset matriisimuunnokset. Oletuksena WebGL piirtää kappaleet ortografisena projektiona, joten perspektiivipiirtoa varten täytyy määritellä erillinen projektiomatriisi (Anttonen & Salminen 2011). Matriisien käsittelyä varten on saatavilla ilmaisia matriisi- ja vektorikirjastoja, jotka helpottavat esimerkiksi projektiomatriisin luomista. Ilmaiseksi jaettavia kirjastoja ovat muun muassa Sylvester (Sylvester) ja glMatrix (GLMatrix). Listauksessa 2.13. on esitetty esimerkki `drawScene()`-funktiosta, joka käyttää hyväkseen glMatrix-kirjastoa.

Listaus 2.13. *DrawScene()-funktion toteutus (muokattu lähteestä Anttonen & Salminen 2011).*

```
function drawScene() {
    gl.clearColor(0.0, 0.0, 0.0, 1.0);
```

```
    gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
    var perspective = mat4.create();
    var modelview = mat4.create();
    mat4.perspective(45.0, 1.0, 0.1, 100.0, perspective);
    mat4.identity(modelview);
    mat4.translate(modelview, [0, 0, -10]);

    gl.bindBuffer(gl.ARRAY_BUFFER, squarePosBuffer);
    gl.vertexAttribPointer(shader.vertexPosAttribute,
                           squarePosBuffer.itemSize,
                           gl.FLOAT, false, 0, 0);

    gl.uniformMatrix4fv(shader.ProjMatrix, false,
                        new Float32Array(perspective));
    gl.uniformMatrix4fv(shader.ModelViewMatrix, false,
                        new Float32Array(modelview));
    gl.drawArrays(gl.TRIANGLE_STRIP, 0,
                  squarePosBuffer.numberOfItems);
}
```

Aluksi funktiossa asetetaan täyttöväri, jolla piirtoalue täytetään ruudun tyhjennyksen yhteydessä. Seuraavaksi `Canvas`-elementti tyhjennetään `gl.clear()`-kutsulla, joka tyhjentää video- ja syvyyspuskurit. Tämän jälkeen määritellään käytettävät matriisit. Projektiomatriisiksi luodaan perspektiiviprojektio ja muunnosmatriisiksi luodaan siirtomatriisi, jossa siirto tapahtuu 10 yksikköä negatiiviseen z-suuntaan. Matriisien määrittelyn jälkeen sidotaan luotu `squarePosBuffer`-puskuri käyttöön ja asetetaan sävyttimen attribuuttiosoitin osoittamaan tähän puskuriin. Lopuksi asetetaan sävyttimen uniformit matriisiosoittimet osoittamaan luotuihin perspektiivi- ja muunnosmatriiseihin ja piirretään geometria ruudulle kutsumalla `gl.drawArrays()`-funktiota. (Anttonen & Salminen 2011.)

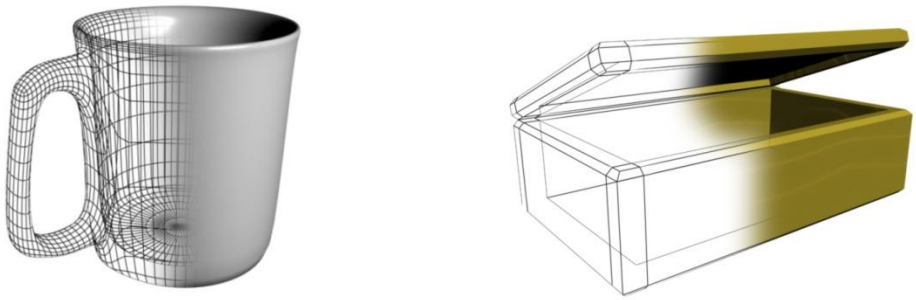
3 TÖRMÄYSTEN HAVAITSEMINEN KOLMIULOTTEISESSA AVARUUDESSA

Tässä luvussa käydään läpi kolmiulotteisen grafiikan esittämisen ja törmäysten havaitsemisen perusteita. Lisäksi luvussa käsitellään kappaleiden ympäröintiä rajauslaatikoilla ja eri rajauslaatikkotyyppien välisiä leikkaustestejä.

3.1 Kolmiulotteiset mallit

Kun esineestä tai asiasta luodaan kolmiulotteinen esitys, siitä saadaan malli, jota sovellukset voivat hyödyntää. Mallia käytetään kuvaamaan kolmiulotteisen avaruuden kappaletta. Mallilla tarkoitetaan jonkin esineen tai asian kolmiulotteista esitystä, kun taas kappaleella tarkoitetaan yhtä mallin geometrian omaavaa esiintymää kolmiulotteisessa avaruudessa. Malli on vain geometrinen esitys, kun taas kappaleella voi olla esimerkiksi sijainti ja nopeus avaruudessa.

Kappaleen kolmiulotteinen malli voidaan luoda esimerkiksi mallinnusohjelman avulla, jolloin mallin geometriaa voidaan muokata helposti. Esimerkkejä näin luoduista malleista on esitetty kuvassa 3.1. Mallin geometria voidaan määritellä myös manuaalisesti ilman mallinnusohjelmaa, mutta se on huomattavasti työläämpää.



Kuva 3.1. *Mallinnusohjelman avulla luotuja kolmiulotteisia malleja.*

Jatkossa keskitytään monikulmioista muodostuviin malleihin, sillä kolmioiden avulla esitetyt mallit ovat nopeampia piirtää verrattuna esimerkiksi matemaattisesti määriteltyihin kappaleisiin. Vaikka kaarevat muodot kärsivät monikulmioiden avulla esitettyinä, on monikulmioiden käyttäminen reaaliaikaisessa piirtämisessä kuitenkin perusteltua, sillä monikulmioiden piirtäminen on helppoa ja nopeaa (Puhakka 2008, s. 50). Nopeus korostuu varsinkin webiin tehtävässä sisällössä, koska selainohjelmoinnissa käytettävän

JavaScript kielen vaatima virtuaalikone on huomattavasti hitaampi verrattuna ohjelman suorittamiseen suoraan käyttäjän koneelta (Taivalsaari et al. 2011).

Taso on keskeinen käsite, kun tarkastellaan kolmiulotteisen avaruuden geometrisia kappaleita. Yksi tapa määritellä taso on valita yksi tason piste **p** sekä kaksi keskenään erisuuntaista vektoria **u** ja **v**, jotka määräävät tason suunnan. Nyt jokainen tason piste **x** toteuttaa yhtälön:

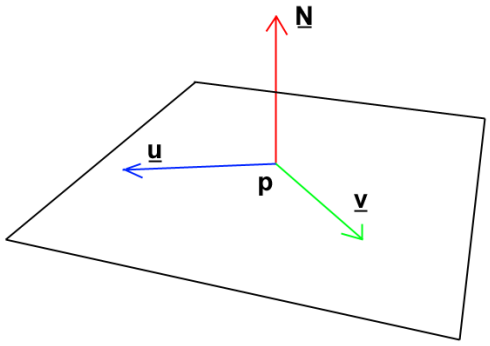
$$\boldsymbol{x} = \boldsymbol{p} + u\underline{\boldsymbol{u}} + v\underline{\boldsymbol{v}} \tag{1}$$

parametrien u ja v joillakin reaaliarvoilla. (Puhakka 2008, s. 41.)

Tasojen yhteydessä puhutaan usein normaalivektorista, joka on yksikäsitteinen suunnan, mutta ei pituuden suhteen, ja kohtisuorassa tasoon nähden. Normaalivektorin avulla siis määritellään, kumpi tason erottamista avaruuden puoliskoista on tason etupuolella. Normaalivektori **N** saadaan tason suuntavektoreiden avulla ristitulona:

$$\underline{\boldsymbol{N}} = \underline{\boldsymbol{u}} \times \underline{\boldsymbol{v}} \tag{2}$$

jolloin siis saadaan suuntavektoreita **u** ja **v** ja niiden määrittelemää tasoa kohtisuorassa oleva vektori. Taso voidaan nyt määritellä yksikäsitteisesti antamalla yksi tason piste **p** sekä tason jokin normaalivektori **N**. (Puhakka 2008, s. 41–42.) Tasoa ja sen normaalivektoria on havainnollistettu kuvassa 3.2.



Kuva 3.2. *Taso, sen määrittelevät vektorit* ***u*** *ja* ***v****, sekä tason normaalivektori* ***N*** *(muokattu lähteestä Puhakka 2008, s. 41).*

Törmäysten havaitsemisessa on usein selvitettävä kahden kappaleen välinen törmäys monikulmion, yleensä kolmion, tarkkuudella. Tähän ongelmaan liittyen on tärkeää osata määrittää, mitkä pisteet kuuluvat annettuun tasoon. Puhakka (2008, s. 42) esittelee yhden tavan suorittaa tällainen laskutoimitus. Yleisesti tasoon kuuluvat ne pisteet, joiden erotusvektori tason pisteestä **r** on kohtisuorassa tason normaalivektoriin nähden:

$$(\boldsymbol{g} - \boldsymbol{r}) \cdot \underline{\boldsymbol{N}} = 0 \;\Rightarrow\; \boldsymbol{g} \cdot \underline{\boldsymbol{N}} - \boldsymbol{r} \cdot \underline{\boldsymbol{N}} = 0. \tag{3}$$

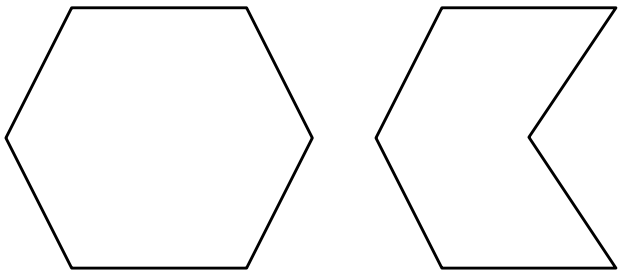
Kun nyt määritellään normaalivektori $\underline{\mathbf{N}} = [A, B, C]^T$ ja merkitään $D = -\mathbf{r} \cdot \underline{\mathbf{N}}$, pätee tason pisteille yhtälö:

$$Ag_x + Bg_y + Cg_z + D = 0 \tag{4}$$

jolloin jos annettu piste $\mathbf{g} = [g_x, g_y, g_z]^T$ toteuttaa tämän yhtälön, piste kuuluu tasoon. (Puhakka 2008, s. 42.)

Monikulmio on kaksiulotteisesta tasosta suljetulla murtoviivalla rajattu alue, ja se määritellään yleensä murtoviivan kulmapisteiden avulla (Puhakka 2008, s. 46). Murtoviiva on kuvio, joka muodostuu useista viivasegmenteistä siten, että edellisen segmentin loppupiste on seuraavan segmentin alkupiste (Puhakka 2008, s. 40). Monikulmion kulmapisteistä yleensä oletetaan, että sama piste ei esiinny kulmapisteiden jonossa peräkkäin, ja että mitkään kolme peräkkäistä kulmapistettä eivät ole samalla suoralla. Jos kyseessä on kolmiulotteisen avaruuden monikulmio, asetetaan yleensä myös vaatimus, että kaikkien kulmapisteiden tulisi sijaita samalla tasolla. Kun kaikki edellä esitetyt oletukset pätevät, voidaan monikulmion normaalivektori laskea minkä tahansa kolmen perättäisen kulmapisteen avulla. (Puhakka 2008, s. 46.)

Monikulmio voi olla konveksi, jolloin kaikki monikulmion reunan sisäkulmat ovat alle 180°, tai ei-konveksi, jolloin kulmat voivat olla yli 180°. Kuvassa 3.3. on annettu esimerkki molemmista tapauksista. Konveksin monikulmion erikoistapaus on kolmio, jonka kolme kulmapistettä määrittelevät aina tason olettaen, että kaikki pisteet eivät sijaitse samalla suoralla. (Puhakka 2008, s. 47–48.)



Kuva 3.3. *Konveksi ja ei-konveksi monikulmio.*

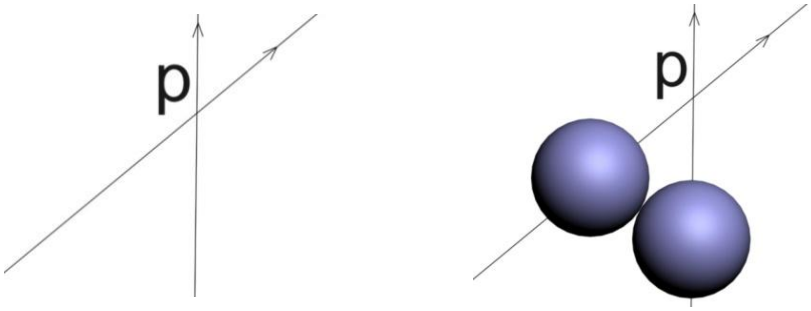
Monitahokas on kolmiulotteisen avaruuden vastine monikulmiolle, ja sen reuna muodostuu äärellisestä joukosta yhteen liitettyjä avaruuden monikulmioita. Näitä reunamonikulmioita kutsutaan tahkoiksi, niiden reunasegmenttejä särmiksi ja särmien kohtauspisteitä kärjiksi. Tahkot muodostavat monikulmioverkon, jota käytetään usein käytännön piirrossa mallintamaan monitahokasta. Monitahokas voidaan määritellä monikulmioiden tapaan konveksiksi, jolloin minkä tahansa kahden monitahokkaaseen kuuluvan pisteen välille piirretty viivasegmentti kuuluu kokonaisuudessaan monitahokkaaseen. (Puhakka 2008, s. 54–55.)

Törmäystarkastelun näkökulmasta se, onko kappale konveksi vai ei-konveksi, voi vaikuttaa merkittävästi algoritmin toteutukseen. Pääsääntöisesti algoritmit käsittelevät konvekseja kappaleita, sillä niiden väliset leikkaustestit voidaan suorittaa nopeasti (Liu et al. 2008). Jos syötteenä on kuitenkin mahdollista saada myös ei-konvekseja kappaleita, pilkotaan tällaiset kappaleet usein konvekseihin osiin (Jiménez et al. 2001). Tällöin ei-konveksien kappaleiden väliseen törmäysten havaitsemiseen voidaan soveltaa konvekseille kappaleille kehitettyjä menetelmiä.

3.2 Törmäysten havaitsemisen peruselementit

Törmäysten havaitseminen on prosessi, jossa tutkitaan, leikkaavatko kappaleet toisiaan avaruudessa. Kappaleiden välisiä törmäyksiä voidaan selvittää perinteisten leikkaustestien avulla, mutta tarkkojen leikkaustestien lukumäärää voidaan kuitenkin vähentää käyttämällä törmäysten havaitsemisessa yksinkertaisempia geometrisia muotoja.

Yksinkertaisin törmäystarkastuksen muoto kahden liikkuvan kappaleen välillä on tapaus, jossa pistemäiset kappaleet liikkuvat suuntavektoreidensa suuntaisesti äärettömän pitkälle. Tällöin niiden välinen törmäys voi tapahtua ainoastaan suuntavektoreiden leikkauspisteessä, jos kappaleet saavuttavat leikkauspisteen samalla ajanhetkellä. Tämä testaus on kuitenkin riittämätön, jos kappaleiden geometriat käsittävät pistettä laajemman alueen. Edellä kuvattuja tilanteita on havainnollistettu kuvassa 3.4.



Kuva 3.4. *Pistemäisten kappaleiden törmäys voi tapahtua ainoastaan suuntavektoreiden leikkauspisteessä* ***p****, mutta geometrialtaan monimutkaisemmilla kappaleilla tilanne ei ole näin yksiselitteinen.*

Kun kaksi ei-pistemäistä kappaletta leikkaavat toisiaan kolmessa ulottuvuudessa, tulee tällainen tilanne voida selvittää laskennallisesti. Seuraavaksi käydään läpi leikkaustestien ja törmäysten havaitsemisen perusteita, jotta voidaan hahmottaa myöhemmin läpikäytävien menetelmien toimintaperiaatteita.

3.2.1 Leikkaustestit kolmessa ulottuvuudessa

Eräs tärkeimmistä kolmiulotteisen avaruuden leikkaustesteistä on suoran ja tason välinen leikkaus, sillä monet monikulmioita ja -tahokkaita koskevat ongelmat voidaan palauttaa koskemaan niiden perusosia, eli suoraa ja tasoa (Puhakka 2008, s. 143). Tavalli-

sesti törmäysten havaitsemisessa tutkitaan lopulta kolmioiden välisiä leikkauksia, joten on yksinkertaisinta tutkia, leikkaako toisen kolmion yksikään reunasegmentti toisen kolmion määrittämää tasoa.

Jos määritellään kolmion reunasegmentti S(t) parametriesityksen avulla käyttämällä reunapistettä **p** sekä suuntavektoria **pq**, saadaan esitysmuodoksi:

$$S(t) = \boldsymbol{p} + t(\underline{\boldsymbol{pq}}). \tag{5}$$

Näin esitetyn suoran ja kaavan (4) avulla esitetyn tason välinen leikkauspiste voidaan selvittää ratkaisemalla muuttuja t yhtälöstä:

$$t = -\frac{\underline{\boldsymbol{N}} \cdot \boldsymbol{p} + D}{\underline{\boldsymbol{N}} \cdot \underline{\boldsymbol{pq}}} \tag{6}$$

jolloin sijoittamalla saatu arvo suoran yhtälöön saadaan leikkauspisteen koordinaatit. (Puhakka 2008, s. 144–145.) Tämän jälkeen on vielä tutkittava, kuuluuko löydetty piste kolmion rajaamalle alueelle. Tämä voidaan tehdä esimerkiksi kiertämällä kolmion viivasegmentit vastapäivään ja tutkimalla, että saatu piste jää jokaisen segmentin vasemmalle puolelle (Puhakka 2008, s. 122).

3.2.2 Törmäysten havaitseminen

Yksinkertainen testi, joka selvittää kaikkien mahdollisesti liikkuvien kappaleiden väliset törmäykset, on esitetty pseudokoodina listauksessa 3.1. Pseudokoodissa oletetaan, että ympäristöön kuuluu N mahdollisesti liikkuvaa kappaletta A_1, A_2, ... , A_{N-1}, A_N ja tarkoituksena on selvittää niiden väliset törmäykset aikavälillä [0, t’] siten, että aikamuuttujaa t kasvatetaan joka kierroksella Δt:n verran. (Hubbard 1993.)

Listaus 3.1. *Yksinkertainen algoritmi ympäristön kappaleiden väliseen törmäysten havaitsemiseen (muokattu lähteestä Hubbard 1993).*

```
for t := 0 to t’ in steps of Δt
    for each Ai ∈ { A1, A2, ... , AN-1, AN }
        move Ai to its position at time t
        for each Aj ∈ { Ai+1, Ai+2, ... , AN-1, AN }
            move Aj to its position at time t
            if (surfaces of Ai and Aj penetrate)
                then a collision occurs at time t
```

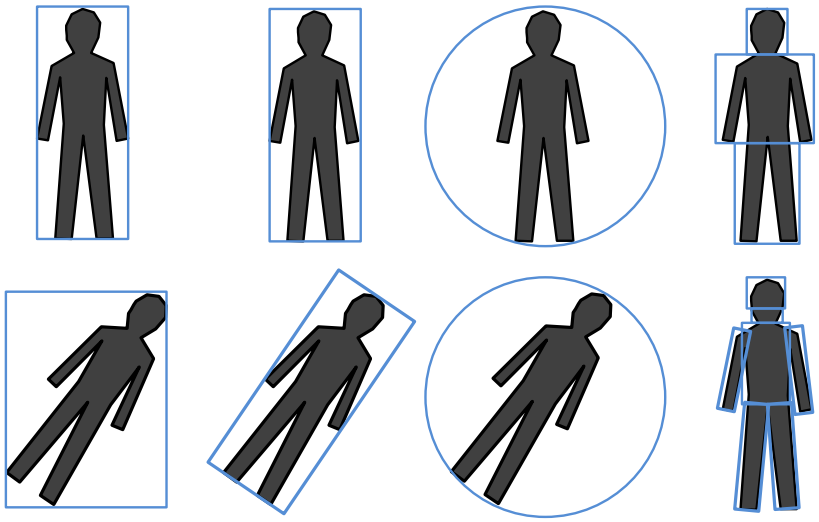
On huomattava, että kasvattamalla Δt:tä saadaan algoritmin suoritusta nopeutettua, mutta pienemmällä arvolla algoritmi havaitsee törmäykset tarkemmin (Hubbard 1993). Algoritmin heikkous on se, että siinä suoritetaan paljon turhaa laskentaa esimerkiksi sellaisten kappaleiden välillä, jotka ovat suuren välimatkan päässä toisistaan. Tällaiset tilanteet on tunnistettava, jotta tarkkaa leikkaustestiä ei tarvitse suorittaa kaikille ava-

ruuden kappaleille. Tarkka leikkaustestaus rajataan siis koskemaan kappaleita, jotka ovat lähellä toisiaan.

Kahden kappaleen A_i ja A_j pintojen välinen leikkaus voidaan toteuttaa vertaamalla näiden kappaleiden kaikkia kolmioita toisiinsa (Quinlan 1994). Tämä voidaan suorittaa yksinkertaisesti tutkimalla leikkaako kummankaan kappaleen minkään kolmion yksikään särmä toisen kappaleen minkään kolmion pintaa. Jos yksikin tällainen leikkaustesti antaa positiivisen tuloksen, tiedetään, että kappaleet leikkaavat toisiaan. Muussa tapauksessa kappaleet ovat irti toisistaan. (Schneider & Eberly 2003, s. 540.) Vaikka tämän tyyppinen toteutus on tarkka, se on kuitenkin laskennallisesti raskas (Beneš & Villanueva 2005). Varsinkin reaaliaikaisissa ja interaktiivisissa sovelluksissa tämän menetelmän käyttäminen ei ole järkevää, sillä kappaleiden väliset törmäykset tulisi laskea nopeasti, jotta sovellus ei menetä interaktiivista luonnettaan (Cohen et al. 1995).

3.2.3 Rajauslaatikot

Törmäystarkastuksessa suoritettavan laskennan yksinkertaistamiseksi voidaan käyttää yksinkertaisempia geometrisia muotoja, rajauslaatikoita, joiden sisään varsinainen kappale ympäröidään. Esimerkkejä tällaisista muodosta on esitetty kuvassa 3.5.



Kuva 3.5. *Pääakselien suuntainen rajauslaatikko, käännetty rajauslaatikko, rajauspallo sekä käännetyistä rajauslaatikoista muodostettu rajauslaatikkohierarkia.*

Käytettävän rajauslaatikon valintaan vaikuttaa Gottschalk et al. (1996) mukaan kaksi vaatimusta. Rajauslaatikon tulisi rajata kappale mahdollisimman tiiviisti, ja laatikoiden välinen etäisyyslaskenta tulisi olla nopea suorittaa. Parempi istuvuus vähentää kappaleen ympärillä olevaa tyhjää tilaa, jolloin rajauslaatikoiden, ja sitä kautta varsinaisten kappaleiden väliset törmäykset vähenevät. Seuraavaksi esitellään kolme yleisesti käytettyä rajauslaatikkoa, jotka ovat pääakselien suuntainen rajauslaatikko, käännetty rajauslaatikko ja rajauspallo.

Pääakselien suuntainen rajauslaatikko muodostuu siten, että laatikon särmät ovat avaruuden pääakselien suuntaisesti. Tällainen laatikko on helpointa määritellä sen koordinaattien minimi- ja maksimiarvojen avulla (Puhakka 2008, s. 148). Kun kolmiulotteista kappaletta pyöritetään, täytyy rajauslaatikko joissakin tilanteissa laskea uudelleen, jotta kappale pysyisi laatikon sisällä. Jos kyseessä on ohut ja pitkä kappale, joka ei ole pääakselin suuntainen, ei sitä kuitenkaan voida rajata tiiviisti (Gottschalk et al. 1996). Pääakselien suuntaisen rajauslaatikon erityistapaus on pääakselien suuntainen kuutio, joka voidaan määritellä yksinkertaisesti keskipisteen ja säteen avulla. Jos rajattava kappale on muodoltaan pallomainen, rajaa kuutio kappaleen hyvin (Cohen et al. 1995).

Jotkin törmäysten havaitsemiseen kehitetyistä menetelmistä käyttävät hyväkseen käännettyjä rajauslaatikoita. Niiden erona pääakselien suuntaiseen rajauslaatikkoon on laatikon kääntyminen kappaleen mukana, mikä puolestaan voi pienentää huomattavasti rajattavan kappaleen ympärille jäävää turhaa tilaa. Koska käännetyt rajauslaatikot rajaavat kappaleen tiiviimmin, niiden välisiä törmäyksiä voi tapahtua harvemmin, millä voi olla suuri vaikutus etenkin suuria kappalemääriä käsiteltäessä (Jiménez et al. 2001). Käännettyjen rajauslaatikoiden välinen etäisyyslaskenta on kuitenkin hankalampaa suorittaa verrattuna pääakselien suuntaiseen rajauslaatikkoon. (Gottschalk et al. 1996).

Pallo on törmäystarkastelun näkökulmasta yksinkertaisin geometrinen muoto, sillä sen määrittelemiseksi riittää tietää pallon keskipiste ja säde. Yleisesti ottaen pallo ei kuitenkaan rajaa kappaletta kovinkaan tarkasti (Beneš & Villanueva 2005). Kahden pallon välinen etäisyyslaskenta on kuitenkin helppo suorittaa, ja lisäksi pallon sisään rajattua kappaletta voidaan pyörittää ilman, että pallo täytyy laskea uudelleen.

3.2.4 Rajauslaatikkohierarkia

Kokonaisen kappaleen approksimointi yhdellä rajauslaatikolla on nopeaa, mutta ei välttämättä tarkkaa. Jos kappale on monimutkainen, kuten esimerkiksi ihminen, törmäysten havaitsemista voidaan tarkentaa rakentamalla rajauslaatikoista hierarkia. Suuremmat rajauslaatikot sisältävät tällöin pienempiä rajauslaatikoita, jotka voivat sisältää vielä pienempiä rajauslaatikoita ja niin edelleen (Puhakka 2008, s. 306).

Rajauslaatikot muodostavat siis puumaisen rakenteen, jossa juurisolmuna on koko kappaleen ympäröivä rajauslaatikko. Tämän juurisolmun lapsina on solmuja, jotka käsittävät alkuperäistä rajauslaatikkoa tarkemman esityksen kappaleesta pienempien rajauslaatikoiden avulla. Nämä lapsisolmut voivat sisältää uusia lapsisolmuja, jotka edelleen tarkentavat kappaleen rajausta. Törmäystarkastusta tehtäessä verrataan aluksi juurisolmua, ja jos törmäys havaitaan, suoritetaan tarkastus lapsisolmuille. Jos törmäys havaitaan niistä toisen solmun kanssa, voidaan toinen solmu jättää testaamatta ja nopeuttaa näin laskentaa

Algoritmin toimintaan vaikuttaa huomattavasti se, mistä rajauslaatikoista hierarkia rakennetaan. Mitä paremmin rajauslaatikko rajaa kappaletta jokaisella hierarkiatasolla, sitä tarkemmin voidaan määrittää varsinaiset törmäävät monikulmiot hierarkian alimmalla tasolla. Tämä tarkoittaa monissa tilanteissa väärien positiivisten vähenemistä.

Curtis et al. (2008) määrittelevät väärät positiiviset kolmiopareiksi, joiden välinen leikkaus testataan, mutta jotka eivät kuitenkaan todellisuudessa leikkaa toisiaan.

Quinlanin (1994) mukaan rajauslaatikkohierarkia voidaan muodostaa monissa sovelluksissa alkulaskentavaiheessa, jolloin jokaiselle avaruuden kappaleelle muodostetaan hierarkkinen rajauslaatikkoesitys ennen varsinaista sovelluksen toiminnallisuutta. Gottschalk et al. (1996) esittävät, että kaksi yleisimmin käytettyä tapaa puun muodostamiselle ovat ylhäältä alas ja alhaalta ylös. Ylhäältä alas liikuttaessa puu muodostetaan jakamalla isompia osia rekursiivisesti pienemmiksi siten, että jäljelle jäävät lehtisolmut, jotka rajaavat esimerkiksi kappaleen yksittäisiä monikulmioita. Alhaalta ylös liikuttaessa lähdetään liikkeelle lehtisolmuista, joita yhdistetään rekursiivisesti suuremmiksi ryhmiksi ja päädytään lopulta koko kappaleen rajaavaan rajauslaatikkoon.

3.3 Rajauslaatikoiden väliset leikkaustestit

Leikkaustestin suorittaminen kahden rajauslaatikon välillä riippuu rajauslaatikoiden tyypeistä. Jos laatikot ovat tyypeiltään samanlaiset, voidaan leikkaustestissä hyödyntää tyypille ominaisia piirteitä. Tämä tarkoittaa yleensä suoraviivaisempaa laskentaa verrattuna tilanteeseen, jossa rajauslaatikoiden tyypit ovat erilaiset.

Kuten alakohdassa 3.2.2. mainittiin, on suositeltavaa rajata tarkka leikkaustestaus koskemaan kappaleita, jotka ovat lähellä toisiaan. Yksi törmäävien kappaleiden etsimiseen kehitetty menetelmä on pyyhkäise ja karsi (engl. sweep and prune). Tässä menetelmässä jokainen avaruudessa oleva kappale on rajattu pääakselien suuntaisella rajauslaatikolla, joiden avulla selvitetään ne kappaleet, joille suoritetaan tarkempi törmäystarkastus.

Pyyhkäise ja karsi -menetelmässä rajauslaatikoiden välinen leikkaustestaus aloitetaan projisoimalla jokainen rajauslaatikko kaikille kolmelle pääakselille. Näin muodostuu jokaista akselia vastaava lista, joka sisältää kaikkien rajauslaatikoiden alku- ja loppupisteet kyseisellä akselilla. Kun nämä listat järjestetään, saadaan tieto kaikista leikkaavista rajauslaatikoista. Näitä rajauslaatikoita vastaavat kappaleet lisätään aktiivisten parien listaan ja välitetään edelleen tarkalle törmäystarkastusalgoritmille. Menetelmä käyttää listojen järjestämiseen insertion sort -algoritmia, jolloin yleisessä tapauksessa listan järjestämiseen kuluisi aikaa $O(n*\log(n))$, jossa n on kappaleiden lukumäärä. Hyödyntämällä koherenssia listat ovat kuitenkin lähes järjestyksessä kahden peräkkäisen ajanhetken välillä, jolloin järjestäminen voidaan suorittaa odotetussa O(n) ajassa. (Cohen et al. 1995.)

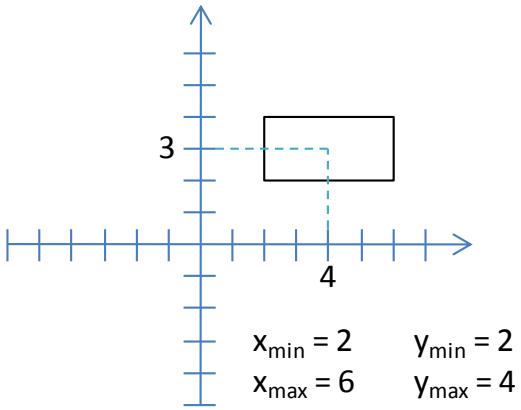
Ajallinen koherenssi tarkoittaa, että ohjelman tila ei muutu huomattavasti kahden peräkkäisen ajanhetken välillä. Tämä pätee siis esimerkiksi tilanteessa, jossa kappaleen paikka muuttuu kahden peräkkäisen kuvan välillä vain vähän. Kappaleen sijainnin pieni muutos tarkoittaa pientä muutosta sen kärkipisteiden koordinaateissa, jolloin kyseessä on geometrinen koherenssi. Jotta koherenssia voidaan hyödyntää, tulee voida olettaa, että kappaleet eivät liiku kahden kuvan välillä pitkiä matkoja. (Cohen et al. 1995.)

3.3.1 Pääakseleiden suuntainen rajauslaatikko

Kahden pääakselien suuntaisen rajauslaatikon välinen leikkaus voidaan havaita tutkimalla rajauslaatikon koordinaattien muodostamia minimi- ja maksimiarvoja. Kolmiulotteisessa tapauksessa yksi pääakseleiden suuntainen rajauslaatikko muodostaa täten kuusi tarkastettavaa arvoa. Jotta kaksi pääakseleiden suuntaista rajauslaatikkoa leikkaavat toisiaan, tulee niiden leikata toisensa kaikilla kolmella pääakselilla. Puhakka (2008, s. 148) esittää, että laatikot leikkaavat toisiaan silloin, kun kaikki seuraavat ehdot pätevät:

$$x^1_{max} \geq x^2_{min} \qquad x^2_{max} \geq x^1_{min}$$
$$y^1_{max} \geq y^2_{min} \qquad y^2_{max} \geq y^1_{min}$$
$$z^1_{max} \geq z^2_{min} \qquad z^2_{max} \geq z^1_{min}.$$

Laskennassa käytettävät koordinaattien minimi- ja maksimiarvot saadaan summaamalla laatikon sijaintiin sen akselikohtaiset minimi- ja maksimiarvot (Cohen et al. 1995). Kuvassa 3.6. on esitetty yhden pääakselien suuntaisen rajauslaatikon muodostamat koordinaattivälit kahdessa ulottuvuudessa.

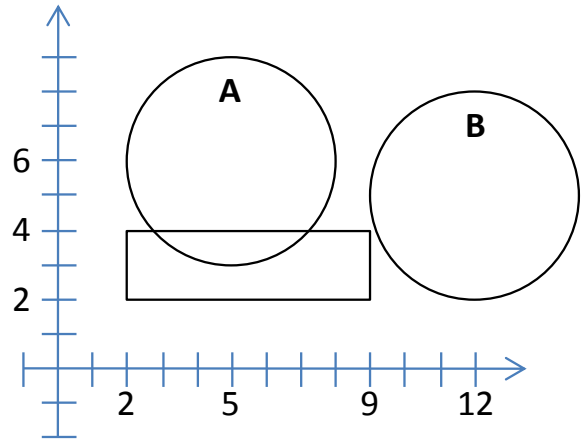


Kuva 3.6. *Yhden pääakseleiden suuntaisen rajauslaatikon muodostamat koordinaattivälit.*

Pääakselien suuntaisen rajauslaatikon ja pallon välinen leikkaus voidaan selvittää tutkimalla pallon keskipisteen etäisyyttä laatikon lähimmästä kohdasta. Jos saatu etäisyys on pienempi kuin pallon säde, kappaleet leikkaavat toisiaan. Tehokas tapa etäisyyden laskemiseen on tarkastella vuorollaan pallon keskipisteen kutakin koordinaattia. Jos keskipiste jää tarkasteltavassa suunnassa laatikon vastaavien sivujen väliin, siirrytään tutkimaan seuraavaa koordinaattia. Jos keskipiste on tarkasteltavassa suunnassa laatikon ulkopuolella, lasketaan kuinka kaukana keskipiste on tässä suunnassa laatikon lähimmästä sivusta. Kun kaikki koordinaatit on käyty läpi, lasketaan saatujen etäisyyksien neliöt yhteen, jolloin tulos on pallon keskipisteen ja laatikon välisen lähimmän etäisyyden neliö. Pallo leikkaa laatikkoa, jos tulos on pienempi tai yhtä suuri kuin pallon säteen

neliö. Sama menetelmä toimii sekä kolmiulotteiselle pallolle että kaksiulotteiselle ympyrälle. Ympyrän tapauksessa käsitellään vain yksi koordinaatti vähemmän. (Puhakka 2008, s. 149–150.)

Kuvassa 3.7. on esitetty pääakseleiden suuntaisen rajauslaatikon ja pallon välinen leikkaustesti kahdessa ulottuvuudessa.



Kuva 3.7. *Pääakseleiden suuntaisen rajauslaatikon ja kahden ympyrän välinen leikkaustesti (muokattu lähteestä Puhakka 2008, s. 150).*

Ympyrän A keskipiste on pisteessä [5, 6] ja ympyrän B pisteessä [12,5]. Molempien ympyröiden A ja B säde on 3. Pääakseleiden suuntainen rajauslaatikko muodostaa puolestaan koordinaattivälit:

$$x_{max} = 9 \qquad y_{max} = 4$$
$$x_{min} = 2 \qquad y_{min} = 2.$$

Ympyrän keskipiste on laatikon rajojen sisäpuolella, kun tarkastellaan ympyrän A x-koordinaattia ja verrataan sitä laatikon rajoihin. Seuraavat ehdot siis pätevät:

$$A_x > x_{min} \quad \text{ja} \quad A_x < x_{max}.$$

Tämän jälkeen tarkastellaan ympyrän y-koordinaattia. Nyt ympyrän keskipiste ei sijaitse laatikon rajojen sisäpuolella, vaan:

$$A_y > y_{min} \quad \text{mutta} \quad A_y > y_{max}.$$

Tässä suunnassa ympyrän keskipisteen lyhin etäisyys laatikon lähimmästä sivusta on:

$$d_y = A_y - y_{max} = 6 - 4 = 2.$$

Leikkaus voidaan nyt selvittää laskemalla neliöt saaduista etäisyyksistä ja vertaamalla saatua arvoa pallon säteen neliöön. Jos saatu etäisyys on pienempi kuin säteen neliö, pallo leikkaa laatikkoa:

$$d_y{}^2 \leq r_A{}^2 = 2^2 \leq 3^2.$$

Pallo A siis leikkaa laatikkoa.

Pallon B keskipiste ei ole laatikon rajojen sisäpuolella kummassakaan suunnassa, joten sekä x- että y-koordinaatin tapauksessa täytyy tutkia ympyrän lyhimpiä etäisyyksiä laatikon sivuihin. Etäisyyksiksi saadaan:

$$d_x = B_x - x_{max} = 12 - 9 = 3$$
$$d_y = B_y - y_{max} = 5 - 4 = 1.$$

Ympyrän leikkaus laatikon kanssa voidaan nyt selvittää laskemalla

$$d_x{}^2 + d_y{}^2 \leq r_B{}^2 = 3^2 + 1^2 \leq 3^2.$$

Pallo B ei siis leikkaa laatikkoa.

3.3.2 Käännetty rajauslaatikko

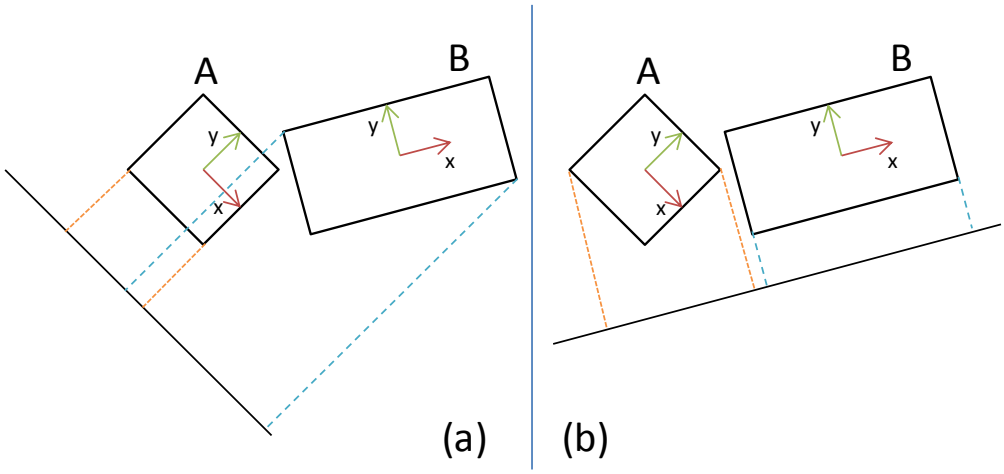
Käännetty rajauslaatikko voidaan käsittää pääakseleiden suuntaisen rajauslaatikon laajennuksena, jossa laatikon koon lisäksi otetaan huomioon laatikon rotaatio. Alakohdassa 3.3.1. mainittu koordinaattien minimi- ja maksimiarvojen tarkastelu ei siis riitä tilanteessa, jossa rajauslaatikkoa käännetään kolmiulotteisessa avaruudessa.

Yksinkertainen algoritmi, joka testaa, leikkaavatko kaksi käännettyä rajauslaatikkoa toisiaan, voidaan toteuttaa käyttämällä ainoastaan särmä-tahko-testejä. Yksi rajauslaatikko muodostuu 12 särmästä ja kuudesta tahkosta. Kun testataan, leikkaako ensimmäisen rajauslaatikon yksikään särmä toisen laatikon yhtäkään tahkoa, täytyy suorittaa $12 * 6$ särmä-tahko-testiä. Testaus täytyy suorittaa myös toiseen suuntaan, jolloin kahden rajauslaatikon välille muodostuu yhteensä 144 särmä-tahko-testiä. Gottschalk et al. (1996) tuovat esille, että tällainen testaus on raskas suorittaa, ja se ei ole käytännöllinen reaaliaikaisissa sovelluksissa.

Yksi tapa yksinkertaistaa kahden käännetyn rajauslaatikon välistä leikkaustestiä on projisoida rajauslaatikot jollekin avaruuden akselille. Tällöin jokaisen laatikon projektio muodostaa kahden päätepisteen rajaaman välin kyseisellä akselilla. Jos kahden rajauslaatikon määrittämät välit eivät mene päällekkäin projisointiakselilla, kutsutaan akselia erottavaksi akseliksi (Gottschalk et al. 1996).

Yhden käännetyn rajauslaatikon särmät muodostavat kolme mahdollista erottavaa akselia, jolloin kahden rajauslaatikon tapauksessa särmien muodostamia mahdollisia erottavia akseleita on kuusi. Kahden laatikon särmien välisistä ristituloista saadaan vielä yhdeksän mahdollista akselia, jolloin kahden rajauslaatikon välille muodostuu yhteensä 15 mahdollista erottavaa akselia. Jos laatikot törmäävät, niiden muodostamien projektioiden täytyy leikata toisensa kaikilla 15 akselilla. Jos taas laatikot ovat erillään, tulee vähintään yhden näistä akseleista olla erottava akseli. Tämä mahdollistaa algoritmin lopettamisen aikaisessa vaiheessa, sillä testaus kahden laatikon välillä voidaan lopettaa heti, kun yksikin erottava akseli löytyy. (Gottschalk et al. 1996.)

Jos kyseessä on pääakselien suuntaisen ja käännetyn rajauslaatikon välinen leikkaustesti, voidaan myös tähän tilanteeseen soveltaa erottavan akselin teoreemaa. Tällöin pääakselien suuntainen rajauslaatikko käsitellään käännettynä rajauslaatikkona, jonka käännösmatriisina on yksikkömatriisi. Kuvassa 3.8. on esitetty erottavan akselin teoreema käytännössä kahdessa ulottuvuudessa. Kuvan tilanteessa (a) erottavana akselina tutkitaan laatikon A lokaalia x-akselia ja tilanteessa (b) vastaavasti laatikon B lokaalia x-akselia.



Kuva 3.8. *Erottavan akselin teoreema kahdessa ulottuvuudessa (muokattu lähteestä Gottschalk et al. 1996).*

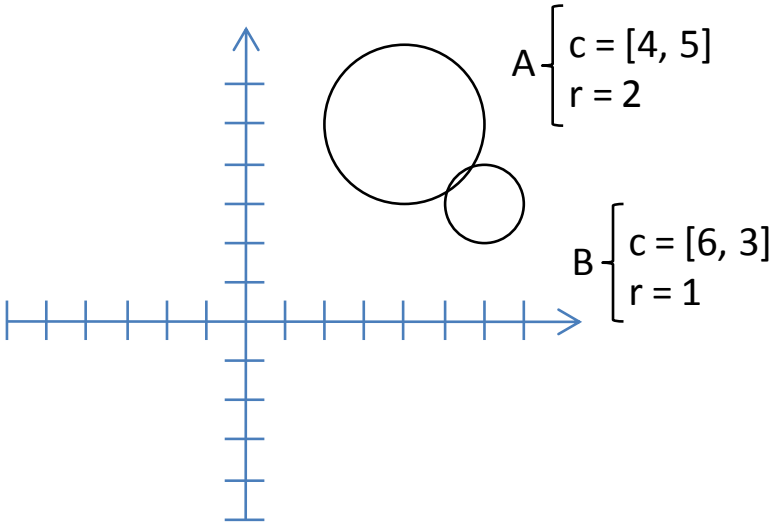
Laatikot projisoidaan tutkittavalle akselille ja jos kappaleiden projektiot leikkaavat, voivat myös itse kappaleet leikata toisiaan. Kaksiulotteisessa tapauksessa riittää tutkia käännetyiden rajauslaatikoiden lokaalit akselit, eli kuvan 3.8. tapauksessa laatikon A x- ja y-akselit ja laatikon B x- ja y-akselit. Kuvan 3.8. (a) tilanteessa laatikoiden projektiot leikkaavat, joten testaamista on jatkettava. Kuvassa 3.8. (b) on esitetty tilanne, jossa testaaminen on edennyt vaiheeseen, jossa tutkitaan laatikon B x-akselia. Laatikoiden muodostamat projektiot tälle akselille eivät leikkaa, joten kappaleetkaan eivät leikkaa toisiaan. Testaaminen voidaan nyt lopettaa, eikä mahdollisesti tutkimatta jääneitä akseleita tarvitse testata, koska yhdenkin erottavan akselin löytäminen riittää.

3.3.3 Pallo

Kuten alakohdassa 3.2.3. mainittiin, kahden pallon välinen leikkaustesti on huomattavasti helpompi suorittaa verrattuna pääakseleiden suuntaiseen tai käännettyyn rajauslaatikkoon. Jos pallojen etäisyys toisistaan on korkeintaan sama kuin säteiden summa, pallot leikkaavat toisiaan. Kahden pallon välinen etäisyysvektori kolmiulotteisessa avaruudessa saadaan pallojen keskipisteiden sijaintivektoreiden erotuksena, jonka jälkeen pallojen välinen etäisyys saadaan laskemalla etäisyysvektorin pituus. Kaksi kolmiulotteisen avaruuden palloa P1 ja P2 leikkaa toisiaan jos seuraava ehto pätee:

$$\sqrt{(x_{P1} - x_{P2})^2 + (y_{P1} - y_{P2})^2 + (z_{P1} - z_{P2})^2} \leq r_{P1} + r_{P2}. \quad (7)$$

Menetelmää soveltuu myös kaksiulotteiseen tapaukseen, jossa pallojen sijaan käsitellään ympyröitä. Kaksiulotteisessa tapauksessa keskipisteiden koordinaateista pudotetaan vain z-koordinaatti pois. Kuvassa 3.9. on esitetty kahden ympyrän A ja B välinen leikkaus.



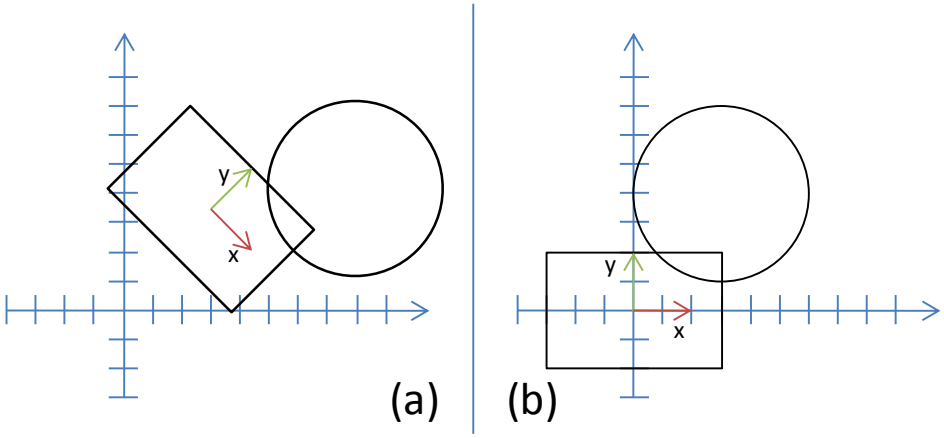
Kuva 3.9. *Kahden ympyrän, A ja B, välinen leikkaustesti.*

Ympyröiden leikkaus voidaan siis selvittää sijoittamalla arvot kaavaan (7):

$$\sqrt{(x_{P1} - x_{P2})^2 + (y_{P1} - y_{P2})^2} \leq r_{P1} + r_{P2} \quad =>$$
$$\sqrt{(4-6)^2 + (5-3)^2} \leq 2 + 1 \quad =>$$
$$\sqrt{(-2)^2 + (2)^2} \leq 3 \quad =>$$
$$\sqrt{8} \leq 3 \quad =>$$
$$2{,}8284 \ldots \leq 3$$

jolloin tuloksena saadaan, että ympyrät leikkaavat toisiaan, mikä näkyy selvästi myös kuvassa 3.9.

Käännetyn rajauslaatikon ja pallon välinen leikkaustesti voidaan suorittaa käyttämällä hyväksi pääakselien suuntaisen rajauslaatikon ja pallon välistä testausmenetelmää. Aluksi pallo siirretään käännetyn rajauslaatikon lokaaliin koordinaatistoon, jolloin käännettyä rajauslaatikkoa voidaan käsitellä pääakseleiden suuntaisena rajauslaatikkona. Pallon ja laatikon välinen lyhin etäisyys voidaan nyt selvittää käyttämällä samaa tekniikkaa kuin pääakseleiden suuntaisen rajauslaatikon ja pallon tapauksessa. Kuvassa 3.10. on havainnollistettu käännetyn rajauslaatikon ja pallon välistä leikkaustestiä.



Kuva 3.10. *Käännetyn rajauslaatikon ja pallon välinen leikkaustesti vaiheittain kahdessa ulottuvuudessa.*

Ennen varsinaisen leikkaustestin suorittamista ympyrä siirretään käännetyn rajauslaatikon koordinaatistoon. Kun siirto on tehty, voidaan käännettyä rajauslaatikkoa käsitellä pääakseleiden suuntaisena rajauslaatikkona, joka muodostaa koordinaattivälit:

$$x_{max} = 3 \qquad y_{max} = 2$$
$$x_{min} = -3 \qquad y_{min} = -2.$$

Ympyrän keskipiste siirtyy muunnoksessa pisteestä [8,4] pisteeseen [3,4]. Muunnos ei vaikuta ympyrän säteeseen, joten säteenä säilyy arvo 3. Nyt tilanne voidaan käsitellä pääakseleiden suuntaisen rajauslaatikon ja ympyrän välisellä leikkaustestillä.

4 KIRJASTO TÖRMÄYSTEN HAVAITSEMISEEN SELAIMESSA

Tässä luvussa esitellään luodun törmäystarkastelukirjaston tärkeimmät ohjelmistovaatimukset ja niiden täyttämiseksi suunniteltu arkkitehtuuri. Luvun tarkoituksena on nostaa esille suunnittelun ja toteutuksen yksityiskohtia ja perustella tehtyjä ratkaisuja. Luvussa käydään lisäksi läpi arkkitehtuurista tunnistettuja suunnittelumalleja. Lopuksi annetaan johdatus kirjaston käyttämiseen ja listataan kirjaston käyttöön liittyviä riippuvuuksia ja rajoituksia. Jatkossa luokat ja oliot on erotettu tekstistä käyttämällä lihavoitua fonttia.

4.1 Yleiskuvaus

Ohjelmistoarkkitehtuurin tehtävänä on ottaa kantaa keskeisiin ohjelmistoratkaisuihin, jotka voivat koskea useita eri ohjelmiston osa-alueita. Näitä osa-alueita ovat esimerkiksi ohjelmiston jakaminen osiin, osien välinen kommunikointitapa, toiminnallisuuden sijoittelu eri järjestelmän osiin, varautuminen tulevaisuuden tarpeisiin ja uudelleenkäytettävyys. Arkkitehtuuri määrittelee siis järjestelmän ytimen, joka pysyy olennaisilta osiltaan samana toteutuksen ja ylläpidon aikana. (Koskimies & Mikkonen 2005, s. 19.)

Arkkitehtuurin pienimpiä rakenteita ovat komponentit, joiden koolle ei ole asetettu mitään yleisiä rajoituksia. Komponentti voi olla esimerkiksi pieni, olion kaltainen yksikkö, joka tarjoaa muutamia yksinkertaisia palveluita. Olennainen osa ohjelmistoarkkitehtuuria ovat rajapinnat, koska ne määräävät tavat, joilla komponentit kommunikoivat keskenään. (Koskimies & Mikkonen 2005, s. 54–58.)

Rajapinnat ovat yksi ohjelmistotekniikan tärkeimmistä periaatteista ja niiden tarkoituksena on erottaa se, mitä halutaan saada aikaan, ja miten tämä tapahtuu. Tällä pyritään siihen, että palvelun käyttäjä ei riipu palvelun tuottavasta komponentista vaan ainoastaan palvelusta abstraktina käsitteenä. Yksi tai useampi komponentti voi puolestaan toteuttaa rajapinnan, eli antaa palvelulle käytännön toteutuksen. (Koskimies & Mikkonen 2005, s. 57–58.)

Arkkitehtuurin keskeisenä pyrkimyksenä on vähentää ja selkeyttää riippuvuuksia, mikä käytännössä usein tarkoittaa erilaisten epäsuoruuksien, kuten rajapintojen ja välittäjäluokkien, lisäämistä (Koskimies & Mikkonen 2005, s. 75). Toisaalta arkkitehtuuri voidaan nähdä kokoelmana korkean tason suunnitteluratkaisuja, jotka pyrkivät tiettyjen vaatimusten ja niistä seuraavien ongelmien ratkaisuun (Koskimies & Mikkonen 2005, s. 22).

4.1.1 Tärkeimmät ohjelmistovaatimukset

Toteutetun törmäystarkastelukirjaston pyrkimyksenä on mahdollistaa törmäysten havaitseminen useissa selainsovelluksissa sovelluksen tyypistä riippumatta. Tämä lähtökohta asettaa kirjastolle vaatimuksia, joihin kirjaston toteutuksen on vastattava. Kirjaston tärkeimpinä vaatimuksina pidettiin suorituskykyä, käytön helppoutta, siirrettävyyttä ja laajennettavuutta.

Kirjaston suorituskyvyllä tarkoitetaan kirjaston törmäysten havaitsemiseen käyttämää aikaa. Hyvä suorituskyky siis vaatii, että kirjasto suorittaa törmäysten havaitsemisen nopeasti, jotta sovelluksen interaktiivinen ja reaaliaikainen luonne säilyy.

Käytön helppoudella pyrittiin siihen, että kirjasto on mahdollisimman helposti liitettävissä sitä käyttävään sovellukseen ja että kirjaston käyttäminen on suoraviivaista. Kirjaston tarjoama rajapinta pidettiin yksinkertaisena, jotta kirjasto on helppo ottaa käyttöön.

Siirrettävyydellä tarkoitetaan, että kirjasto toimii mahdollisimman irrallaan sitä käyttävästä sovelluksesta, jolloin kirjaston päivittyminen tai sisäisen toiminnallisuuden muutos ei aiheuta muutoksia sitä käyttävään sovellukseen. Kirjaston toiminnassa tähän pyrittiin kiinnittämällä huomiota siihen, että kirjaston suorittama törmäystarkastelu tekee mahdollisimman vähän oletuksia kirjastoa käyttävästä sovelluksesta. Siirrettävyydellä pyrittiin myös takaamaan, että kirjasto toimii useiden eri ympäristöjen kanssa.

Hyvällä laajennettavuudella pyrittiin siihen, että esimerkiksi uusia rajauslaatikoita voidaan lisätä kirjastoon helposti ja niiden käyttäminen ei vaadi suuria muutoksia kirjastoon tai sitä käyttävään sovellukseen. Kirjaston sisäisessä rakenteessa pyrittiin eriyttämään törmäystarkastelussa tarvittava yleinen logiikka ja kappalekohtainen leikkaustestien suorittaminen toisistaan. Näin voitiin parantaa kirjaston laajennettavuutta, koska kirjaston perustoiminta ei muutu uusia rajauslaatikoita lisättäessä.

4.1.2 Kerrosarkkitehtuuri

Kohdejärjestelmän hahmottamisessa voidaan käyttää apuna arkkitehtuurityylejä, jotka toimivat toteutuksen rakenteen selityksenä, mutta myös ryhmittely- ja hahmotustekniikkana. Yksi tärkeimmistä arkkitehtuurityyleistä on kerrosarkkitehtuuri, jota voidaan käyttää lähes minkä tahansa järjestelmän kuvaamisessa. (Koskimies & Mikkonen 2005, s. 125.)

Kerrosarkkitehtuuri koostuu tasoista, jotka on järjestetty nousevasti jonkin abstrahointi periaatteen mukaan (esimerkki kuvassa 4.1.). Usein abstrahointiperiaate tarkoittaa skaalaa laite-ihminen, jossa laite-pään kerrokset tarjoavat laitetta ja käyttöjärjestelmää lähellä olevia toimintoja, kun taas ihmistä lähellä olevat kerrokset keskittyvät graafiseen käyttöliittymään. (Koskimies & Mikkonen 2005, s. 126.)

Kuva 4.1. *Kerrosarkkitehtuuri (muokattu lähteestä Koskimies & Mikkonen 2005, s. 128).*

Kerrosarkkitehtuurin perusajatuksena on, että tietyllä tasolla oleva komponentti tai palvelu toteutetaan käyttämällä alemman kerroksen tarjoamia komponentteja tai palveluja. Joissakin tilanteissa palvelukutsu voi ohittaa kerroksia kulkiessaan ylhäältä alas. Syynä tähän on tehokkuuden parantaminen, koska usein palvelu löytyy tehokkaampana alemmilta kerroksilta. (Koskimies & Mikkonen 2005, s. 126.)

Kuvassa 4.2. on esitetty luodun törmäystarkastelukirjaston suunniteltu sijoittuminen eräässä kerrosarkkitehtuurissa.

Kuva 4.2. *Kirjaston sijoittuminen kerrosarkkitehtuurissa.*

Kirjaston toiminnassa on lähdetty siitä, että kirjasto tarjoaa palveluja ainoastaan ylemmille kerroksille, eikä yhteyksiä kirjaston ja sitä mahdollisesti alempien kerrosten välille tarvita. Kerrosarkkitehtuurin yleinen ongelma on tehokkuushäviö, kun palvelua joudutaan siirtämään alaspäin kerroksissa, jolloin toiminnasta tulee epäsuoraa (Koskimies & Mikkonen 2005, s. 131). Kirjaston toiminta on toteutettu siten, että kirjasto voidaan käsittää yhtenä matalan tason arkkitehtuurikerroksen osana. Kirjaston suunnittelussa pyrittiin hyödyntämään suunnittelumalleja, joiden avulla saatiin selkeytettyä kirjaston rakennetta ja ratkaistua joitakin kirjastolle asetettujen ohjelmistovaatimusten tuomia ongelmia.

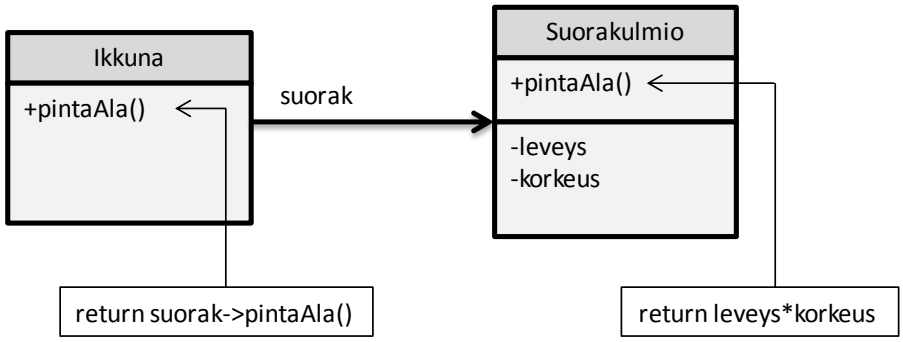
4.2 Suunnittelumallit

Haikala & Märijärvi (2006, s. 365) mukaan suunnittelumallien idea perustuu havaintoon, jossa samantapaiset ratkaisut luokkarakenteen ja luokkien yhteistyön suunnittelussa näyttävät toistuvan. Voidaan siis sanoa, että suunnittelumallit ovat tunnettujen ja käytännössä hyväksi havaittujen ratkaisujen kuvauksia yleisiin ohjelmistojen suunnittelussa ilmeneviin ongelmiin tietyissä tilanteissa. Suunnittelumallien ottaminen käyttöön on helppoa, koska ne eivät edellytä tiettyä teknologiaa, suunnittelumenetelmää tai ohjelmointikieltä. (Koskimies & Mikkonen 2005, s. 101–102.)

Suunnittelumallin keskeiset osat ovat ongelma, ongelmayhteys ja ratkaisu. Ongelman tulee olla yleinen suunnitteluongelma, joka ei edellytä esimerkiksi tiettyä ohjelmointikieltä. Ongelmayhteys kertoo, millaisissa tilanteissa suunnittelumalli on sovellettavissa. Suunnittelumallin ratkaisun tulee olla yleinen, ja sen on oltava kuvattavissa esimerkiksi UML:n avulla. (Koskimies & Mikkonen 2005, s. 105.)

4.2.1 Kutsun siirtäminen

Kutsun siirtäminen ei ole suoranainen suunnittelumalli, mutta se on laajasti käytetty menetelmä ja moni suunnittelumalli käyttää apunaan kutsun siirtämistä (Gamma et al. 1995 s. 20). Kutsun siirtämisen avulla voidaan muuttaa kutsumuotoa ja piilottaa kutsun varsinainen suorittaja. Esimerkki kutsun siirtämisestä on esitetty kuvassa 4.3.

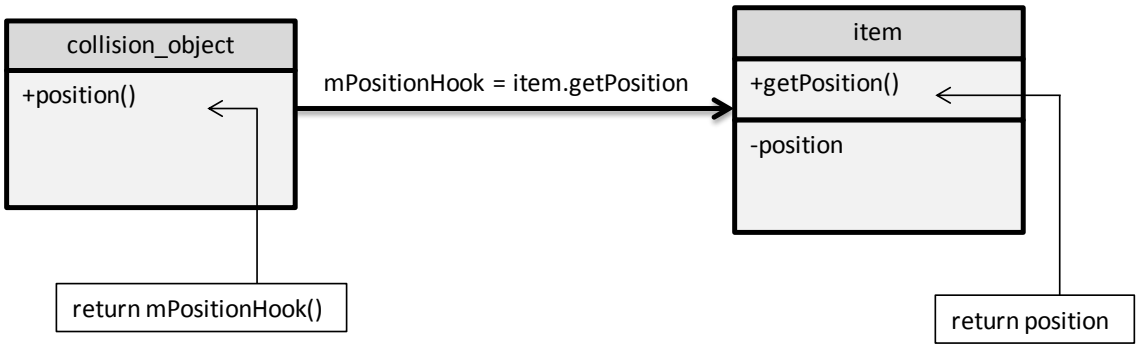


Kuva 4.3. *Kutsun siirtäminen (muokattu lähteestä Gamma et al. 1995, s. 20).*

Kuvassa **Ikkuna**-luokka tarjoaa `pintaAla()`-funktion, joka palauttaa ikkunan sen hetkisen pinta-alan. **Ikkuna**-luokkaa ei kuitenkaan periytetä **Suorakulmio**-luokasta, vaan **Ikkuna**-luokka omistaa instanssin **Suorakulmio**-luokasta. Kun **Ikkuna**-oliolta kysytään pinta-alaa, ei olio laske sitä omien tietojensa avulla, vaan siirtää kutsun **Suorakulmio**-oliolle ja palauttaa saadun tuloksen. (Gamma et al. 1995, s. 20.)

Kirjaston toteutuksessa oli ongelmana se, että kirjaston ei tulisi tehdä oletuksia sitä käyttävästä sovelluksesta. Kutsun siirtämisen avulla voitiin korvata kiinteät rajapintamäärittelyt joustavalla ratkaisulla, joka mahdollistaa eri rajapintojen käyttämisen. Kirjasto määrittelee ainoastaan funktioiden paluuarvojen muodon. Kutsun siirtämistä käytetään **collision_object**-luokan `hook`-funktioissa, joiden avulla **collision_object**-olio

voidaan sitoa avaruuden kappaleeseen. Kuvassa 4.4. on esitetty **collision_object**-olion suorittama kutsun siirtäminen.



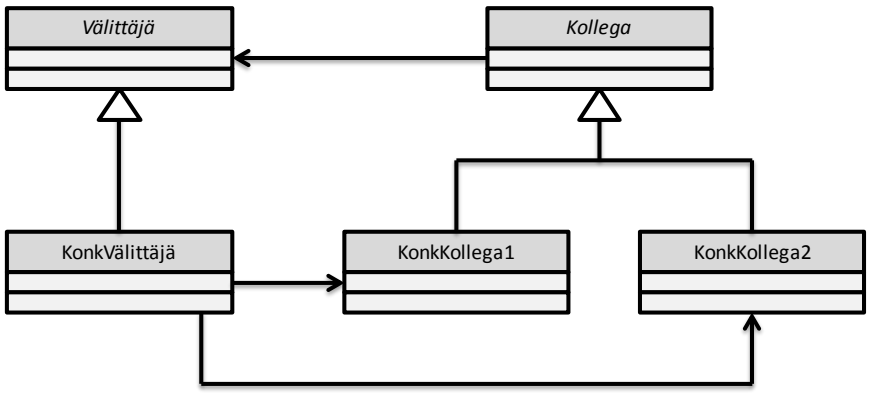
Kuva 4.4. *Kutsun siirtäminen osana* ***collision_object****-olion toteutusta.*

Kun kirjasto pyytää **collision_object**-oliolta sijaintia, siirretään kutsu avaruuden kappaleen `getPosition()`-funktiolle ja palautetaan tämän funktion tulos. Kutsun siirtäminen on tehokas tapa koostaa olioita ja sen avulla ohjelmakoodista on mahdollista tehdä uudelleenkäytettävää ilman periytymistä (Gamma et al. 1995, s. 21).

4.2.2 Välittäjä

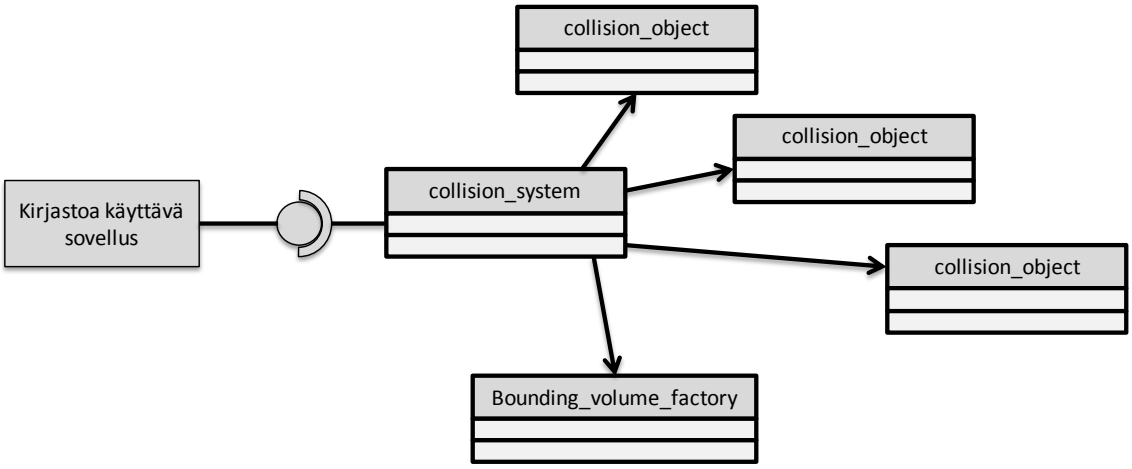
Komponenttien väliset yhteydet vaikeuttavat komponenttien uudelleenkäyttöä, järjestelmän ylläpitoa ja ymmärtämistä, kun komponentit ovat sidottuna mutkikkaaseen vuorovaikutusympäristöön (Koskimies & Mikkonen 2005, s. 79). Ongelman ratkaisuksi voidaan hyödyntää välittäjä-suunnittelumallia, joka toteuttaa osallistujien välisen vuorovaikutuksen. Välittäjä-suunnittelumallin rakenne luokkakaaviona on esitetty kuvassa 4.5.

Välittäjä-suunnittelumalli sopii tilanteisiin, joissa oliot kommunikoivat keskenään määritellysti, mutta monimutkaisesti. Välittäjän käyttö on perusteltua myös tilanteissa, joissa useiden eri olioiden vuorovaikutuksen tulos tulee olla muutettavissa helposti. (Gamma et al. 1995, s. 276.)



Kuva 4.5. *Välittäjä-suunnittelumalli (muokattu lähteestä Gamma et al. 1995, s. 276).*

Kirjaston toteutuksessa ongelmana oli kirjaston tarjoaman rajapinnan mahdollinen hajaantuminen liian monelle eri komponentille ja kirjaston sisäisen rakenteen monimutkaistuminen. Ongelma ratkaistiin tekemällä **collision_system**-luokasta välittäjä, joka sekä tarjoaa kirjaston käyttöön vaadittavan rajapinnan että sisältää kirjaston yleisen logiikan. Kuvassa 4.6. on esitetty **collision_system**-luokan sijoittuminen kirjaston rakenteeseen.



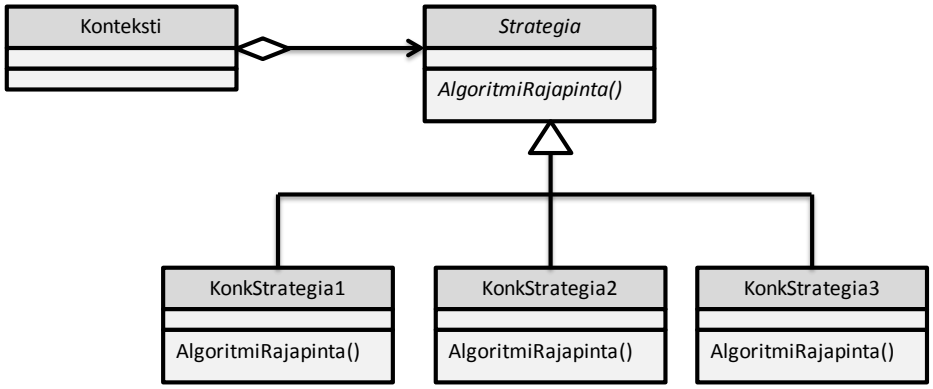
Kuva 4.6. *Kirjastossa välittäjänä toimiva* ***collision_system****-luokka.*

Välittäjän käyttö mahdollistaa sen, että muiden komponenttien vastuita voidaan rajoittaa ja siten poistaa niiden välisiä riippuvuuksia. Kirjaston toteutuksessa kontrollin keskittäminen yhdelle komponentille sopi hyvin kirjaston tarpeisiin. Välittäjänä toimivasta **collision_system**-oliosta ei muodostunut hallitsematon kokonaisuus, vaan komponentin koko säilyi hallittavana kontrollin keskittämisestä huolimatta.

4.2.3 Strategia

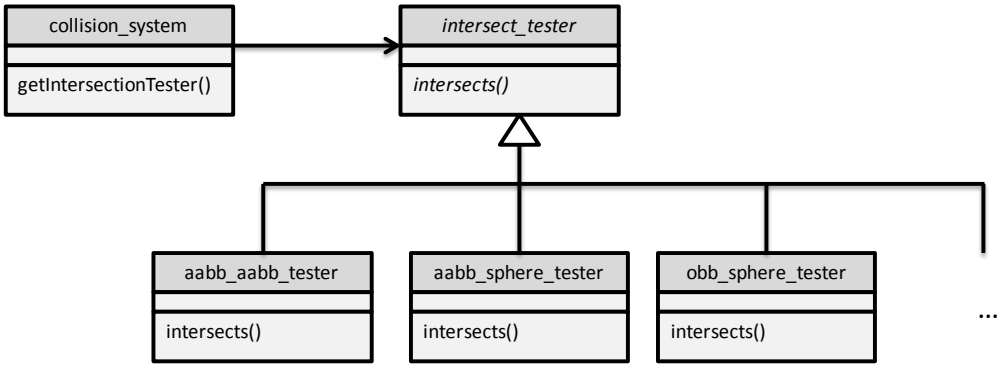
Jos tiedetään, että tietyn algoritmin toiminta voi muuttua ajonaikana, ei eri toiminnallisuuksien kirjoittaminen kiinteäksi osaksi ohjelmakoodia ole mielekästä. Algoritmin eri vaihtoehtojen sijoittaminen kiinteäksi osaksi niitä käyttävää komponenttia kasvattaa komponentin kokoa ja tekee komponentista hankalasti hallittavan. Tähän ongelmatilanteeseen voidaan soveltaa strategia-suunnittelumallia. (Gamma et al. 1995, s. 315.)

Strategia-suunnittelumalli (esitetty luokkakaaviona kuvassa 4.7.) soveltuu tilanteisiin, joissa moni eri luokka poikkeaa toisistaan ainoastaan toteutuksen osalta. Strategia-suunnittelumallin avulla voidaan määritellä yksi luokka, jonka toiminnallisuus on vaihdettavissa. Strategia-suunnittelumallia voidaan käyttää myös tilanteissa, joissa tietystä algoritmista vaaditaan useita eri toteutuksia. (Gamma et al. 1995, s. 316.)



Kuva 4.7. *Strategia-suunnittelumalli (muokattu lähteestä Gamma et al. 1995, s. 316).*

Kirjaston toiminnassa on tärkeää, että kappaleiden leikkauksia tutkiva algoritmi voidaan vaihtaa ajon aikana riippuen tutkittavien kappaleiden tyypeistä. Strategia-suunnittelumallia on hyödynnetty **collision_system**-olion suorittamassa kappaleiden leikkaustestauksessa. Käytettävä leikkaustestin suorittaja valitaan kappaleiden tyypin mukaan, mutta jokainen suorittaja toteuttaa saman rajapinnan. Näin voidaan tutkia kappaleiden leikkaus riippumatta kappaleiden tyypeistä ja oikea leikkausalgoritmi voidaan valita ajon aikana. Kuvassa 4.8. on kuvattu strategia-suunnittelumalli osana **collision_system**-oliota.



Kuva 4.8. *Strategia-suunnittelumalli **collision_system**-olion osana.*

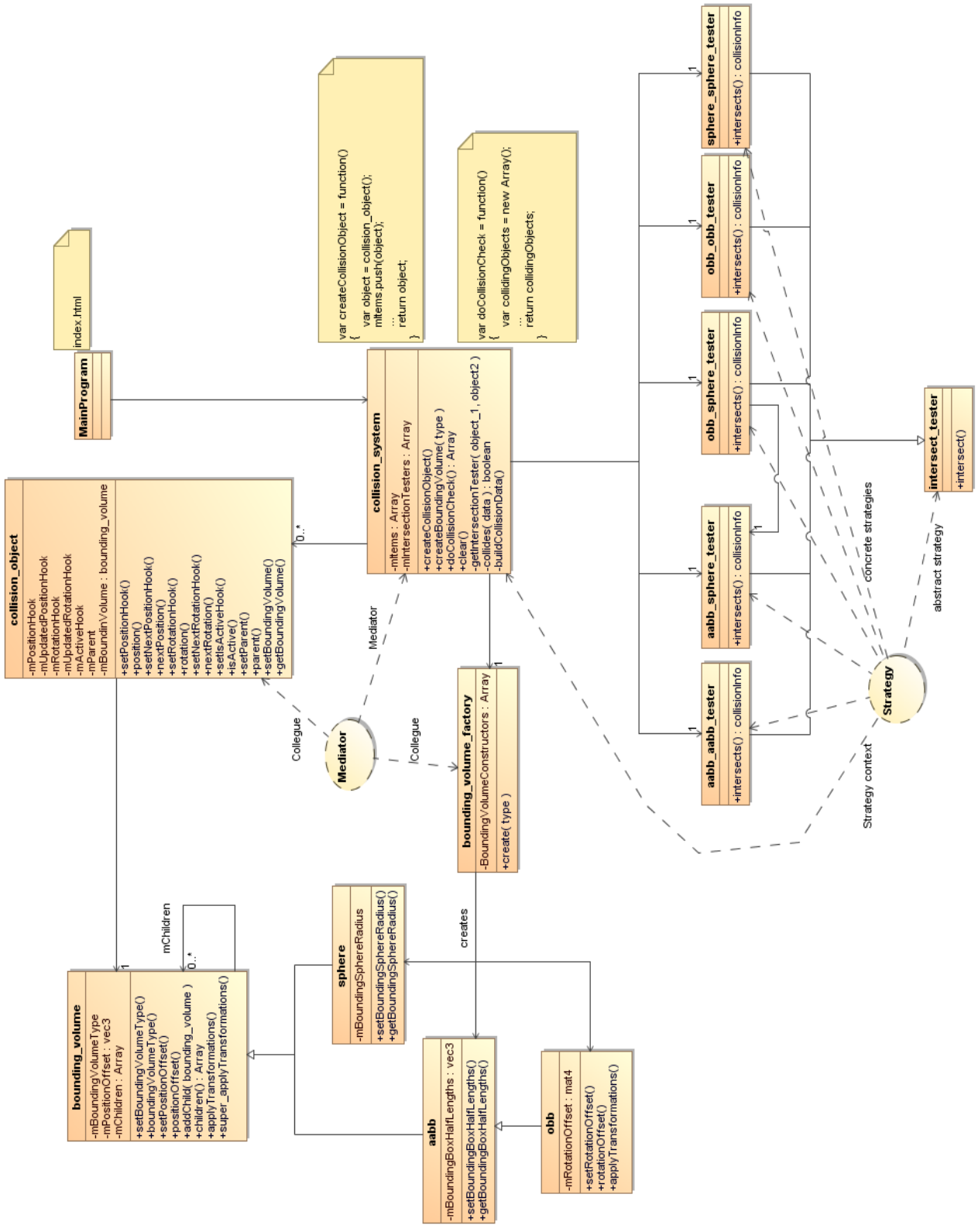
Strategia-suunnittelumalli mahdollistaa hyvän ja helpon laajennettavuuden, koska erilaisten kappaleiden välisiä leikkaustestausalgoritmeja voidaan lisätä helposti kirjastoon. Uusien leikkaustestausalgoritmien tulee toteuttaa ainoastaan **intersect_tester**-rajapinta.

4.3 Arkkitehtuuri

Kirjaston arkkitehtuurissa pyrittiin selkeään kokonaisuuteen, jossa eri komponenttien väliset yhteydet ovat selkeitä. Näin arkkitehtuurista saatiin ylläpidettävä ja hallittava, mikä parantaa kirjaston laajennettavuusmahdollisuuksia.

4.3.1 Oliokaavio

Kirjaston rakenne ja kirjaston toteutukseen liittyvät arkkitehtuuriset ratkaisut osana kirjastoa on esitetty oliokaaviona kuvassa 4.9.



Kuva 4.9. *Kirjaston oliokaavio.*

Kuvassa 4.9. nähdään suunnittelumallien sijoittuminen osaksi kirjaston rakennetta. Välittäjänä toimiva **collision_system**-olio vastaa kirjaston päätoiminnallisuudesta ja on yhteydessä toiminnan kannalta tärkeisiin olioihin. **Collision_system**-olio toimii myös kappaleiden leikkaustestauksen toteuttamiseen käytetyn strategia-suunnittelumallin kontekstina. Myös kirjastoa käyttävä sovellus, joka kuvassa 4.9. on esitetty **MainProgram**-oliona, on yhteydessä **collision_system**-olioon.

Kirjaston toteutuksen lisäksi oliokaaviosta on havaittavissa kirjaston laajennettavuusmahdollisuudet oliotasolla. Kirjastoon voidaan lisätä uusia rajauslaatikko-olioita, kunhan uudet rajauslaatikko-oliot toteuttavat **bounding_volume**-olion määrittelemän rajapinnan. Kirjastoon voidaan lisätä myös uusia leikkaustestaukseen käytettäviä leikkausalgoritmeja, kunhan lisättävät algoritmioliot toteuttavat **intersect_tester**-olion määrittelemän rajapinnan.

4.3.2 Tiedostorakenne

Kirjaston tiedostorakenteessa pyrittiin loogisuuteen, jotta kirjaston käyttö olisi helppoa. Selkeän tiedostorakenteen avulla voitiin myös tukea kirjaston laajennettavuutta, koska tiedostojen selkeä sijoittuminen kirjaston hakemistopuussa auttaa hahmottamaan kirjaston toimintaa kokonaisuutena. Kirjaston tiedostorakenne on esitetty kuvassa 4.10., jossa hakemistojen nimet on esitetty lihavoituina.

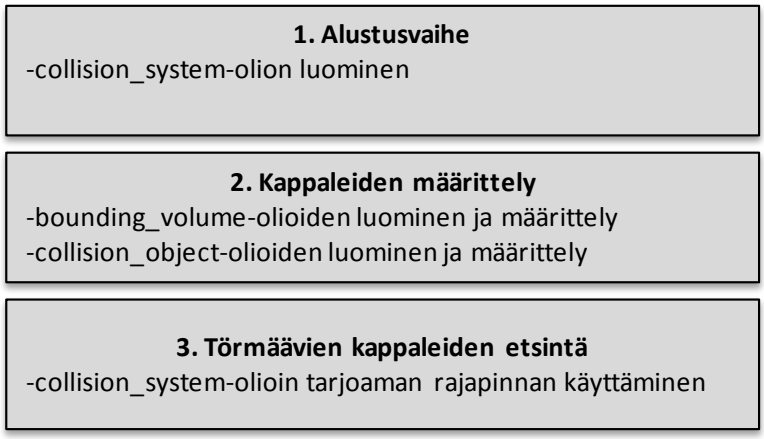
- **Bounding_volumes**
  - aabb.js
  - bounding_volume_factory.js
  - obb.js
  - sphere.js
- **Testers**
  - aabb_aabb.js
  - aabb_sphere.js
  - obb_obb.js
  - obb_sphere.js
  - sphere_sphere.js
- bounding_volume.js
- intersect_tester.js
- collision_object.js
- collision_system.js

Kuva 4.10. *Kirjaston tiedostorakenne.*

Kirjaston juurihakemistoon sijoitettiin tiedostot, jotka sisältävät kirjaston käyttäjälle näkyvien rajapintojen määrittelyt ja kirjaston käytön kannalta merkittävimmät, **collision_system**- ja **collision_object**-luokat. Alihakemistoihin sijoitettiin niiden luokkien toteutukset, jotka eivät suoraan ole näkyvissä kirjaston käyttäjälle. Näitä ovat esimerkiksi yksittäiset leikkausalgoritmit ja rajauslaatikot.

4.4 Kirjaston käyttäminen

Ennen kirjaston käyttämistä on kirjastoon kuuluvat JavaScript-tiedostot lisättävä HTML-dokumentin `<head>`-osioon käyttämällä `script`-elementtejä. Kirjaston käyttäminen voidaan jakaa kolmeen vaiheeseen. Ensimmäisessä vaiheessa luodaan kirjaston käyttämiseen vaadittava olio, toisessa vaiheessa määritellään törmäystarkasteluun kuuluvat kappaleet ja kolmannessa vaiheessa tutkitaan kappaleiden välisiä törmäyksiä ja palautetaan tieto törmäävistä kappaleista. Kuvassa 4.11. on esitetty kirjaston käyttö vaiheittain.



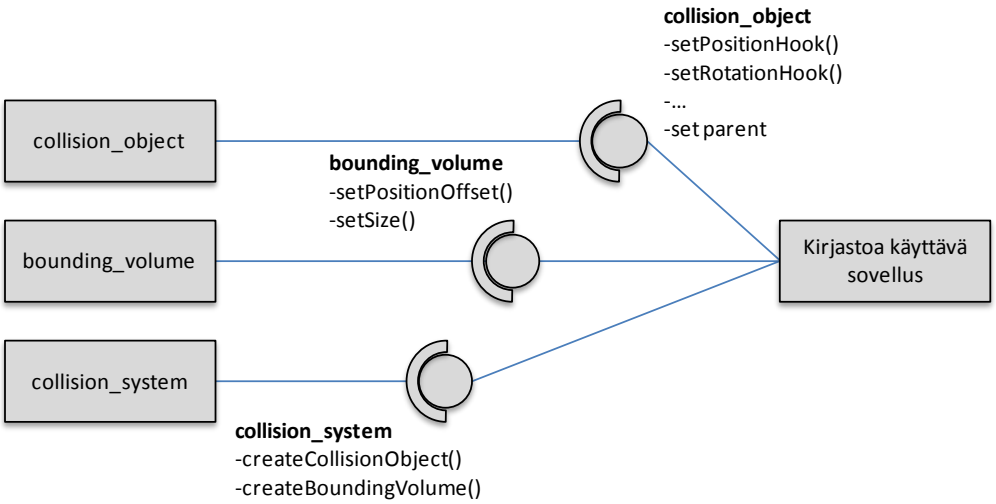
Kuva 4.11. *Kirjaston käytön vaiheet.*

Alustusvaiheessa luodaan **collision_system**-olio, joka toimii välittäjänä kirjaston ja sitä käyttävän sovelluksen välillä ja siten pääasiallisena rajapintana kirjaston käyttämiseksi. Kun olio on luotu, kirjasto on käyttövalmis ja kirjaston käyttäjä siirtyy kappaleiden määrittelyvaiheeseen.

4.4.1 Kappaleiden määrittely

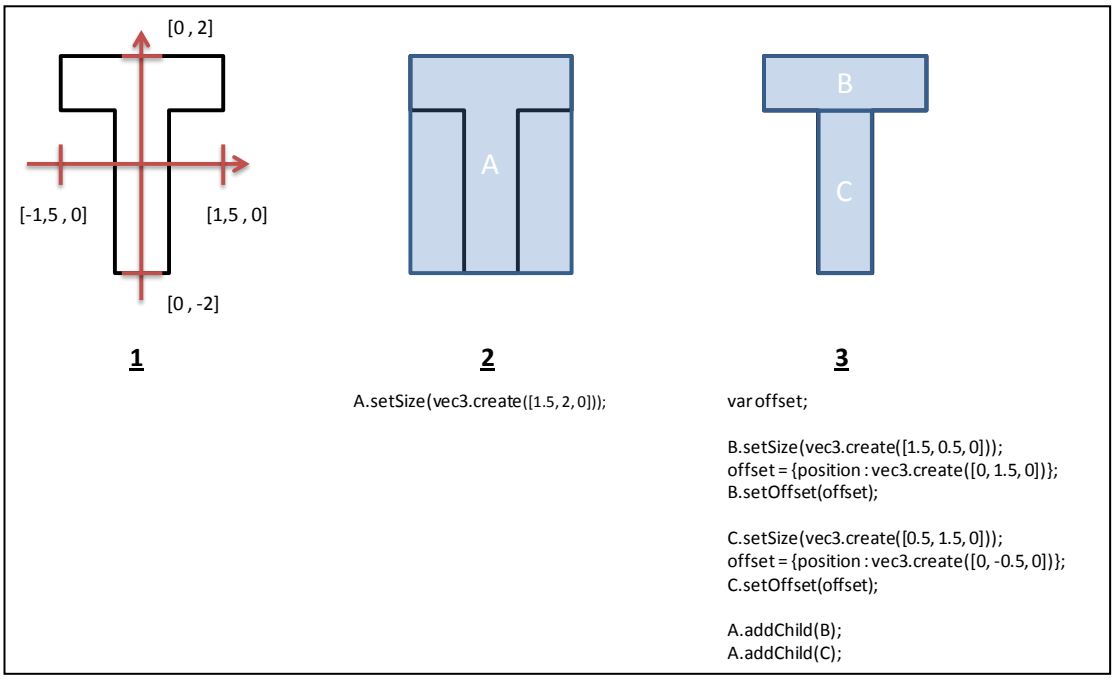
Törmäystarkastelussa käytettävät kappaleet muodostuvat kahdesta osasta: **collision_object**-oliosta ja sen sisältämästä **bounding_volume**-oliosta. Jako voidaan käsittää siten, että **collision_object**-olio sidotaan seuraamaan jotakin avaruuden kappaletta, ja tähän kappaleeseen liittyy yhden **bounding_volume**-luokasta periytetyn olion muodostama rajauslaatikko tai rajauslaatikkohierarkia.

Kappaleiden määrittelyvaiheessa kirjaston käyttäjän kannalta tärkeimmät rajapinnat ovat siis **collision_system**- ja **collision_object**-luokkien rajapinnat sekä **bounding_volume**-luokan määrittelemä rajapinta, jonka rajauslaatikot toteuttavat. Näitä rajapintoja on havainnollistettu kuvassa 4.12. Jatkossa **collision_object**-olioista käytetään nimitystä seurausolio ja **bounding_volume**-luokan aliluokista yleistä nimitystä rajauslaatikko-olio.



Kuva 4.12. *Kappaleiden määrittelyvaiheessa kirjaston käyttäjälle näkyvät rajapinnat.*

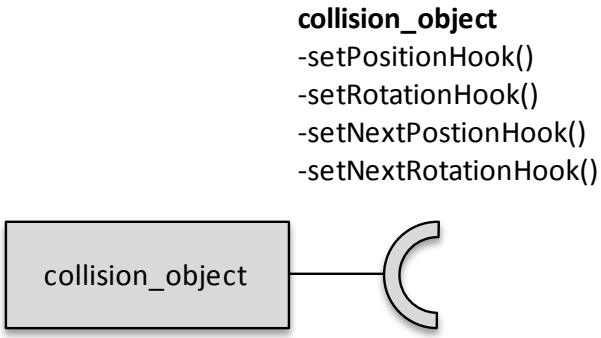
Kappaleita määriteltäessä **collision_system**-rajapinta tarjoaa palvelut sekä rajauslaatikko- että seurausolioiden luomiseksi. Rajauslaatikko-oliota määriteltäessä lähdetään liikkeelle juurirajauslaatikosta, jonka tarkoituksena on ympäröidä koko kappale. Juurirajauslaatikko-oliolle voidaan tämän jälkeen lisätä lapsia, joille voidaan edelleen lisätä omia lapsia ja niin edelleen. Näin voidaan luoda rajauslaatikkohierarkia. Hierarkian luomisessa käytetään apuna rajauslaatikko-olioiden `offset`-metodeja. Niiden avulla esimerkiksi rajauslaatikon sijaintia voidaan poikkeuttaa suhteessa rajauslaatikko-olion isäntäolion sijaintiin. Kuvassa 4.13. on havainnollistettu rajauslaatikkohierarkian luomista.



Kuva 4.13. *Rajauslaatikkohierarkian luominen.*

Hierarkian luomisessa lähdetään liikkeelle koko kappaleen ympäröivästä rajauslaatikosta. Tälle juurirajauslaatikolle lisätään tämän jälkeen lapsirajauslaatikoita, jotka rajaavat kappaletta tarkemmin. Lapsirajauslaatikoiden sijainnit ovat suhteessa ylemmän tason rajauslaatikkoon eivätkä siis itse kappaleeseen. Kun kappaleelle on rakennettu riittävä rajauslaatikko-olio, siirrytään määrittelemään seurausoliota, johon rajauslaatikko-olio sidotaan.

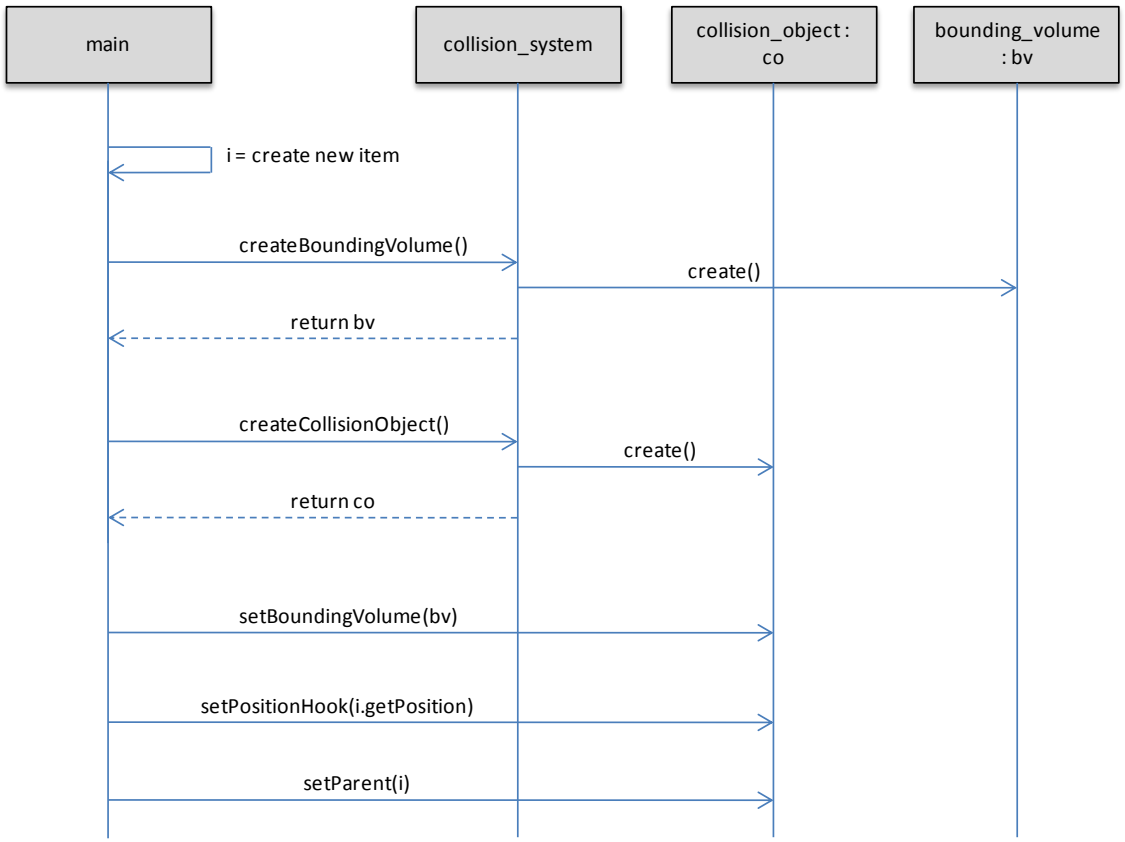
Seurausolion tarkoituksena on tarjota törmäysten havaitsemisprosessille pääsy riittävään informaatioon kohteena olevasta avaruuden kappaleesta. Seurausolio tarjoaa rajapinnan, jonka kautta olio voidaan sitoa avaruuden pisteeseen. Rajapinta perustuu koukkuihin, joiden avulla törmäystarkastelukirjasto pääsee käsiksi tarvitsemaansa tietoon. Kuvassa 4.14. on esitetty seurausolion tarjoama rajapinta sen koukkujen määrittelemiseksi.



Kuva 4.14. *Seurausolion koukkujen määrittelyyn tarjoama rajapinta.*

Rajapinnan `setPositionHook()`- ja `setRotationHook()`-funktioita käytetään asettamaan koukut, joiden kautta saadaan avaruuden kappaleen paikka ja rotaatio ajanhetkellä t_0. Vastaavasti `setNextPositionHook()`- ja `setNextRotationHook()`-funktioita käytetään asettamaan koukut, joiden kautta saadaan kappaleen paikka- ja rotaatiotieto ajanhetkellä t_1. Törmäysten havaitseminen tehdään oletuksena ajanhetkellä t_1, mutta jos tätä koukkua ei ole asetettu, käytetään ajanhetken t_0 koukkuja.

Seurausolion koukkuihin voidaan sijoittaa suoraan haluttu arvo, jolloin törmäystarkastelukirjaston tekemä kutsu palauttaa tämän sijoitetun arvon. Puhtaan arvon lisäksi koukkuun voidaan sijoittaa suoraan jokin avaruuden kappaleen rajapintafunktio. Esimerkiksi kappaleen sijainnin palauttava `get`-funktio. Jos koukkuun sijoitetaan rajapintafunktio, törmäystarkastelukirjaston suorittamat kutsut välitetään seurausolion kautta varsinaiselle avaruuden kappaleelle. Kuvassa 4.15. on esitetty seurausolion määrittely.



Kuva 4.15. *Seurausolion määrittely.*

Seurausoliolle määritellään koukkujen lisäksi vanhempi, jonka kautta päästään käsiksi avaruuden kappaleeseen, jota seurausolio on asetettu seuraamaan. Tätä tietoa voidaan käyttää hyväksi esimerkiksi törmäykseen reagoinnissa. Kun kaikki tarvittavat seuraus- ja rajauslaatikko-oliot on määritelty, kirjaston käytössä siirrytään kolmanteen vaiheeseen.

4.4.2 Törmäävien kappaleiden etsiminen

Kirjaston käytön kolmannessa vaiheessa tutkitaan määriteltyjen kappaleiden välisiä törmäyksiä, mikä käytännössä tarkoittaa törmäävien kappaleiden etsimistä tietyn aikaaskeleen välein. Törmäävien kappaleiden etsintään käytetään **collision_system**-luokan tarjoamaa rajapintaa.

Tietyllä ajanhetkellä törmäävät kappaleet saadaan selvitettyä kutsumalla **collision_system**-olion `doCollisionCheck()`-funktiota. Funktion kutsu tulee suorittaa aina, esimerkiksi jokaisen ruudunpäivityksen yhteydessä, kun halutaan selvittää törmäävät kappaleet. Metodin tuloksena palautetaan lista niistä kappalepareista, jotka törmäävät.

4.5 Rajoitukset ja riippuvuudet

Osassa kirjaston toiminnallisuutta käytettiin kirjaston ulkopuolisia komponentteja, jotka aiheuttavat siten riippuvuuksia kirjaston ulkopuolelle. Kirjaston sisäisessä toiminnassa vaadittavaan vektori- ja matriisilaskentaan käytettiin glMatrix-kirjastoa. Matematiikka-kirjaston käyttämiseen päädyttiin, koska vektori- ja matriisioperaatioiden toteuttaminen kirjaston osana ei ollut perusteltua. Valmis kirjasto tarjoaa optimoidut operaatiot ja selkeän rajapinnan niiden toteuttamiseksi.

Kirjaston toteutuksessa pyrittiin siihen, että kirjasto tekee mahdollisimman vähän oletuksia sitä käyttävästä sovelluksesta. Mahdollisesti törmäävien kappaleiden etsimiseen käytetty pyyhkäise ja karsi -menetelmä hyödyntää toiminnassaan koherenssia. Tämä tarkoittaa, että sovelluksessa käytettyjen kappaleiden ei tulisi liikkua pitkiä matkoja kahden ajanhetken välillä. Kirjaston suorittama tarkka törmäysten havaitseminen on luonteeltaan diskreetti, eli törmäysten havaitseminen suoritetaan ainoastaan aika-askeleen päätepisteessä. Nämä toteutusyksityiskohdat asettavat rajoituksia kappaleiden nopeuksille, jotta törmäykset havaitaan nopeasti ja tarkasti. Käytännön rajoituksia ei listata, koska riittävän aika-askeleen pituus riippuu käytettävän laitteiston laskentatehosta ja sovelluksen toteutuksesta.

Kirjaston toiminta vaatii, että seurausolion määrittelyssä asetettavat `hook`-funktiot palauttavat ennalta määritellyn tyyppiset paluuarvot. Kunkin funktion paluuarvon tyyppi on määritelty funktiokohtaisesti. Kappaleen sijainnin palauttavien `position()`- ja `nextPosition()`-funktioiden tulee palauttaa paluuarvonaan glMatrix-kirjaston määrittelemä kolmiulotteinen vektori (`vec3`). Kappaleen kierron palauttavien `rotation()`- ja `nextRotation()`-funktioiden tulee palauttaa glMatrix-kirjaston määrittelemä 4*4 matriisi (`mat4`).

Havaituista törmäyksistä raportoidaan törmäävät kappaleet sekä leikkaavat rajauslaatikot. Kolmioiden välistä törmäysten havaitsemista ei toteutettu, koska kolmioiden esitysmuoto voi vaihdella sovelluskohtaisesti.

5 KIRJASTON ARVIOINTI

Kirjaston toiminnan testaamiseksi luotiin testisovellus, joka sisältää erilaisia testitilanteita. Sovelluksesta saatujen tulosten perusteella arvioitiin, kuinka hyvin kirjasto vastaa sille asetettuihin ohjelmistovaatimuksiin, ja siten kirjaston soveltuvuutta törmäysten havaitsemiseen.

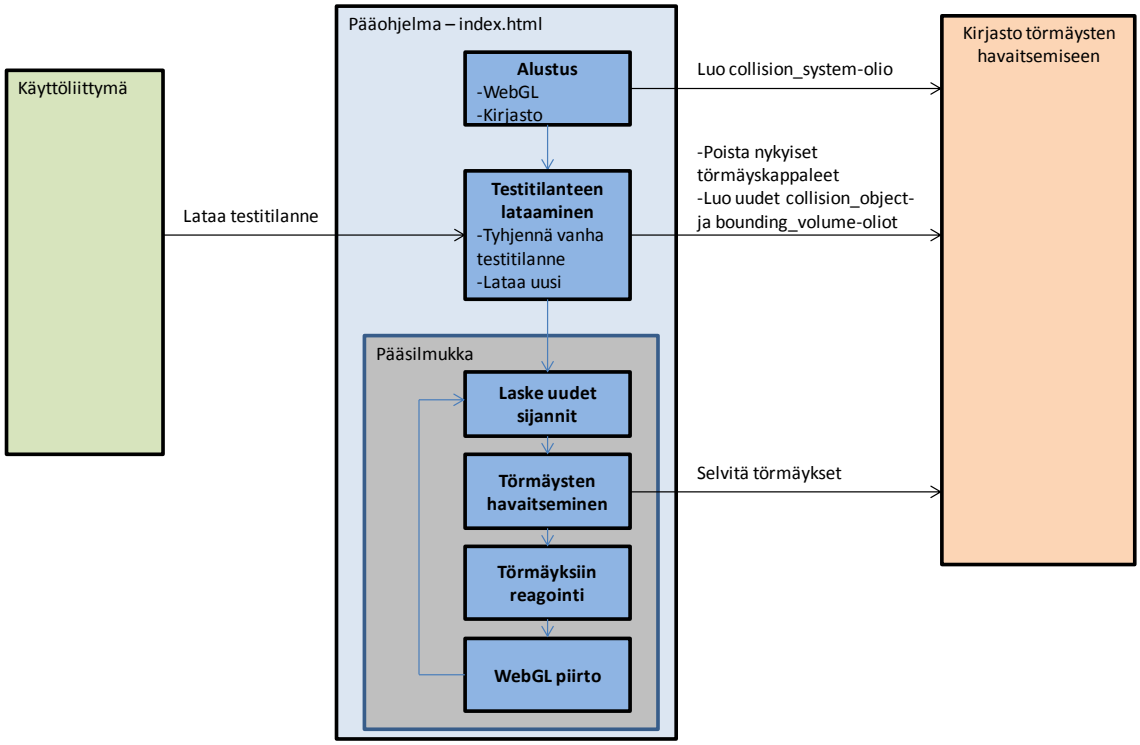
5.1 Testisovelluksen rakenne ja testausympäristö

Kirjaston toimintaa testattiin työpöytäympäristössä yhdellä työasemalla ja testattaviksi selaimiksi pyrittiin valitsemaan yleisesti Windows-ympäristössä käytössä olevia selaimia.. Testausympäristö on esitetty taulukossa 5.1.

Taulukko 5.1. *Testausympäristö.*

Laitteisto	
Suoritin	Intel Core 2 Duo E6750 @ 2.66 GHz
Keskusmuisti	4 GB
Käyttöjärjestelmä	Windows Vista Home Premium, 32 bit, Service Pack 2
Näytönohjain	NVIDIA GeForce GTX 280, 1024 MB
Ajuriversio	266.58
Selain	versio
Mozilla Firefox	6.0.2
Google Chrome	14.0.835.186
Opera	11.50 labs
Internet Explorer	9.0.8112.16421

Testisovellus toteutettiin JavaScript-kielellä ja kolmiulotteisen grafiikan piirto toteutettiin WebGL:llä. Testisovelluksesta tehtiin reaaliaikainen, jotta voitiin arvioida kirjaston soveltuvuutta interaktiivisiin ja reaaliaikaisiin sovelluksiin. Testisovellusta ei ladattu suoritettavaksi palvelimelta, vaan sovellus sijoitettiin testilaitteiston paikalliselle kiintolevylle, josta sovellus suoritettiin. Näin voitiin välttää tiedonsiirrosta johtuvat viiveet, koska sovellusta ja sen osia ei tarvinnut hakea Internetissä sijaitsevalta palvelimelta. Kuvassa 5.1. on esitetty testisovelluksen rakenne.



Kuva 5.1. *Testisovelluksen rakenne.*

Sovelluksen alustusvaiheessa suoritetaan WebGL:n käyttöön tarvittavat alustustoimenpiteet. Alustusvaihe noudattaa aiemmin esitettyjä toimenpiteitä. Alustusvaiheessa luodaan lisäksi törmäysten havaitsemiseen käytettävän kirjaston vaatima **collision_system**-olio. Testisovellukseen toteutettiin käyttöliittymä käyttämällä HTML-standardin mukaisia käyttöliittymäkomponentteja, kuten tekstisyötekenttiä ja painonappeja (HTML 1999). Käyttöliittymän avulla käyttäjä voi vaihtaa suoritettavaa testitilannetta sovelluksen suorituksen aikana. Alustusvaiheen jälkeen sovellus lataa testitilanteen, joka on alustuksessa määritelty ensimmäiseksi.

Testitilanteen lataaminen sisältää kaksi vaihetta. Ensimmäisessä vaiheessa vanha testitilanne irrotetaan sovelluksesta tyhjentämällä vanhaan testitilanteeseen liittyvät tietorakenteet. Törmäysten havaitsemiskirjaston tapauksessa nykyiset tietorakenteet tyhjennetään kutsumalla **collision_system**-olion tarjoaman rajapinnan kautta funktioita `clear()`. Tietorakenteiden tyhjentämisen jälkeen luodaan uuden testitilanteen vaatimat kappaleet ja näille kappaleille vaaditut seuraus- ja rajauslaatikko-oliot. Testitilanteen lataamisen jälkeen käynnistetään sovelluksen pääsilmukka.

Testisovelluksen pääsilmukka koostuu neljästä vaiheesta. Ensimmäisessä vaiheessa avaruuden kappaleille lasketaan uudet sijainnit ja rotaatiot. Näiden uusien tietojen pohjalta suoritetaan törmäysten havaitseminen käyttämällä kirjaston toiminnallisuutta. Käytännössä silmukassa kutsutaan **collision_system**-olion tarjoaman rajapinnan `doCollisionCheck()`-funktiota. Funktion paluuarvona saadaan lista törmäävistä kappalepareista ja jokaiselle kappaleparille leikkaavat rajauslaatikot.

Havaittuihin törmäyksiin reagoidaan muuttamalla törmäävien kappaleiden väritystä. Kappale piirretään puhtaan vihreällä värisävyllä, jos kappale ei törmää yhdenkään toisen kappaleen kanssa. Kappaleen värityksenä käytetään kirkkaan punaista, jos kappale törmää toiseen kappaleeseen. Sovelluksen pääsilmukan viimeisenä vaiheena on WebGL:llä suoritettava piirto, joka piirtää avaruuden kappaleet näytölle käyttämällä törmäysten reagoinnista saatua kappalekohtaista piirtoväriä.

5.2 Testitilanteet

Kirjaston toimintaa testaavien testitilanteiden avulla arvioitiin kirjaston soveltuvuutta törmäysten havaitsemiseen. Testitilanteissa painotettiin kirjaston suorituskykyä eri selaimilla. Törmäysten havaitsemisen tarkkuutta ei testattu, koska laskennallinen törmäysten havaitseminen suoritetaan yleisesti tiedossa olevien menetelmien mukaisesti. Törmäysten havaitsemisen tarkkuus on siten sidoksissa käytettyjen menetelmien tarkkuuteen.

Testitilanteita muodostettiin muuttamalla testissä käytettyjä kappaleiden määriä sekä törmäysten havaitsemisessa käytettävää karsintavaihetta. Karsintavaiheesta tarkasteltiin tilannetta, jossa karsintaa ei suoritettu ennen tarkkaa törmäysten havaitsemista sekä kolmea erilaista tilannetta, joissa karsinta suoritettiin ja kappaleiden maksiminopeutta avaruudessa rajoitettiin. Maksiminopeuden rajoittamisella tutkittiin koherenssin vaikutusta karsintavaiheessa käytetyn pyyhkäise ja karsi -menetelmän toimintaan. Testitilanteista saadun tulostaulukon rakenne on esitetty taulukossa 5.2.

Taulukko 5.2. *Tulostaulukon rakenne.*

	Aika (ms)			
Kappalemäärä	ei karsintaa	Karsinta, v_{max} = 20,0	Karsinta, v_{max} = 5,0	Karsinta, v_{max} = 1,0
10				
100				
1000				

Testitilanteissa kappalemääriksi valittiin 10, 100 ja 1000. Valittuihin kappalemääriin päädyttiin, koska niiden avulla voidaan havainnollisesti tutkia suoritusajan suhdetta kappalemäärään. Jokaiselle kappaleelle valittiin satunnaisesti rajauslaatikko kolmesta vaihtoehdosta. Lisäksi jokainen kappale sijoitettiin satunnaiseen avaruuden pisteeseen **p** siten, että seuraavat ehdot pätevät:

$$p_x > -100 \qquad p_x < 100$$
$$p_y > -100 \qquad p_y < 100$$
$$p_z > -100 \qquad p_z < 100.$$

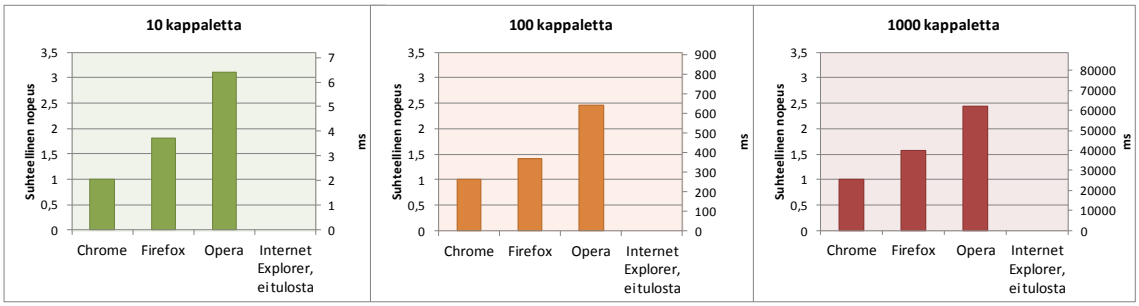
Myös kappaleiden liike rajoitettiin näiden rajojen sisäpuolelle. Ylittäessään avaruuden rajan, kappaleen nopeus käännettiin vastakkaiseksi ylitys suunnan mukaan ja kappaleen paikka siirrettiin avaruuden rajalle.

Testitilanteissa, joissa karsintavaihetta ei käytetty ennen tarkkaa törmäysten havaitsemista, tarkoituksena oli tuottaa mittaustuloksia, joiden avulla voitiin tarkastella selainten välistä suorituskykyä. Karsintavaihetta käytettäessä kappaleiden maksiminopeuksiksi valittiin 20, 5, ja 1. Valittuihin nopeuksiin päädyttiin, koska niiden avulla voitiin testata erisuuruisten nopeuksien muutoksen vaikutusta törmäysten havaitsemisen nopeuteen. Vertaamalla nopeuksien 20 ja 1 välistä muutosta, voidaan tutkia suuren nopeuden muutoksen vaikutusta törmäysten havaitsemisen suorituskykyyn. Pienen nopeuden muutoksen välistä eroa voidaan puolestaan tutkia vertaamalla nopeuksien 5 ja 1 välistä suorituskykyä. Kappalemääriä ja karsintaparametreja muuttamalla testitilanteita luotiin yhteensä 12.

5.3 Mittaustulokset

Selainten välistä suorituskykyä tutkittiin suorittamalla törmäysten havaitsemista ilman karsintavaihetta ja kolmessa eri karsintavaiheen sisältävässä testitilanteessa. Ilman karsintavaihetta suoritettujen testitilanteiden tarkoituksena oli tutkia selainten nopeutta suoritettaessa JavaScript-ohjelmia. Karsintavaiheen sisältävissä testitilanteissa keskityttiin kirjaston toiminnan testaamiseen. Jokainen testitilanne suoritettiin viisi kertaa neljällä eri selaimella, ja jokaisella suorituskerralla törmäysten havaitsemisesta otettiin keskiarvo 20 peräkkäisen päivityskerran välillä. Saadut tulokset suhteutettiin eri kappalemäärien tapauksissa nopeimman selaimen suorituskykyyn.

Testisovellusta ei voitu ajaa Internet Explorer selaimella, koska selain ei tue WebGL-tekniikkaa. Tästä johtuen mitattuja tuloksia saatiin vain Google Chrome-, Mozilla Firefox- ja Opera -selainten suorituksesta. Google Chrome oli jokaisessa testitilanteessa testattavista selaimista nopein. Ilman karsintavaihetta suoritettujen testitilanteiden tulokset on esitetty kuvassa 5.2. Kaikkien testitilanteiden tulokset on esitetty taulukoissa 5.3.–5.5.



Kuva 5.2. *Selainten väliset suhteelliset ja todelliset nopeudet.*

Taulukko 5.3. *Testitilanteista saadut mittaustulokset Google Chrome -selaimella.*

Chrome, ver 14.0.835.186		aika (ms)			
suorituskerta	kappalemäärä	ei karsintaa	karsinta vmax = 20	karsinta vmax = 5	karsinta vmax = 1
1	10	2,1	0,05	0	0
	100	246,5	3,05	3,15	3,1
	1000	25566	50,2	53,55	49,05
2	10	1	0	0	0
	100	269,7	3,25	3,15	3,05
	1000	25061	45,45	49,1	45,95
3	10	2,7	0,05	0,05	0
	100	266,5	3,15	3,05	3,1
	1000	25361,5	59,25	48,95	47,8
4	10	2,7	0	0	0,05
	100	261,2	4,2	3,25	3,05
	1000	25662	59,25	46,45	41,9
5	10	1,8	0	0	0
	100	262	3,05	3,15	3,1
	1000	25693,66	60,2	47,95	40,4
keskiarvo	10	2,06	0,02	0,01	0,01
	100	261,18	3,34	3,15	3,08
	1000	25468,83	54,87	49,2	45,02

Taulukko 5.4. *Testitilanteista saadut mittaustulokset Mozilla Firefox -selaimella.*

Firefox, ver 6.0.2		aika (ms)			
suorituskerta	kappalemäärä	ei karsintaa	karsinta vmax = 20	karsinta vmax = 5	karsinta vmax = 1
1	10	3,35	0,65	0,65	0,55
	100	378,8	5,45	5,3	5,4
	1000	42989	132,35	78	66,25
2	10	4,05	0,5	0,55	0,45
	100	356,9	5,45	5,9	5,45
	1000	39593	136,4	77,8	66,15
3	10	3,45	0,55	0,5	0,35
	100	372,4	5,55	5,1	5,15
	1000	39313	130,1	79,25	66,4
4	10	3,95	0,65	0,5	0,5
	100	373,45	5,65	5,15	5,35
	1000	39033	133,3	78,35	66,3
5	10	3,85	0,55	0,5	0,3
	100	366,25	5,7	5,45	5,3
	1000	39495	129,2	79,55	65,9
keskiarvo	10	3,73	0,58	0,54	0,43
	100	369,56	5,56	5,38	5,33
	1000	40084,6	132,27	78,59	66,2

Taulukko 5.5. *Testitilanteista saadut mittaustulokset Opera -selaimella.*

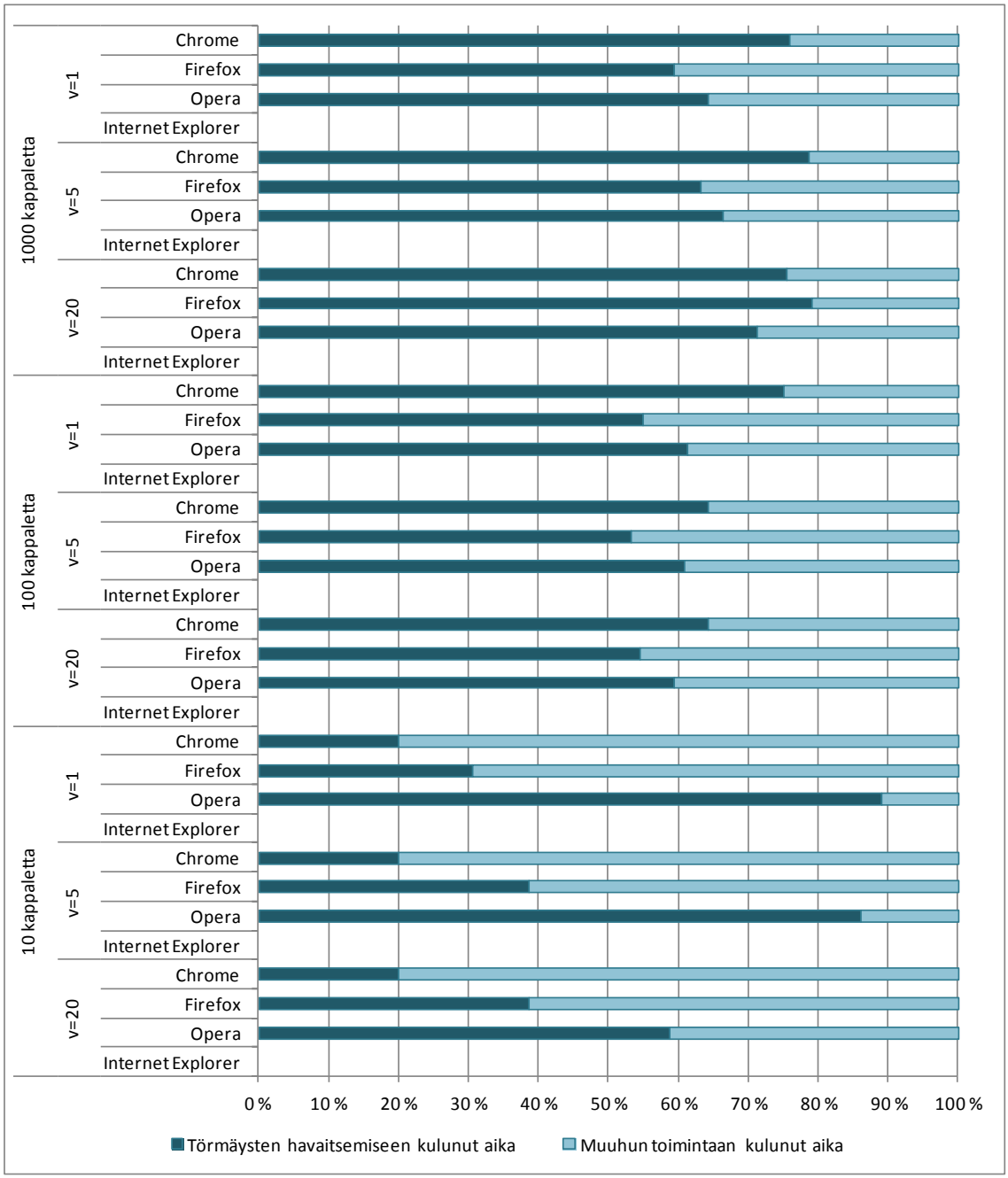
Opera, ver 11.50 labs		aika (ms)			
suorituskerta	kappalemäärä	ei karsintaa	karsinta vmax = 20	karsinta vmax = 5	karsinta vmax = 1
1	10	7,2	1	1	0,95
	100	621,25	6,1	6	6,15
	1000	61387	101,95	77,6	73,55
2	10	5,25	0,9	1	1
	100	639,35	6	6,35	6,05
	1000	61376	99,75	76,55	70,25
3	10	6,95	0,95	1	0,95
	100	621,8	6,05	6,15	6,6
	1000	62026,5	95,15	76,9	71,55
4	10	6,1	1	1	1
	100	660,25	6,05	6,2	6
	1000	62666	96	76,4	71,75
5	10	6,45	1	0,95	1
	100	669,3	6	6,05	6,05
	1000	62225,33	96,4	76,65	70,85
keskiarvo	10	6,39	0,97	0,99	0,98
	100	642,39	6,04	6,15	6,17
	1000	61936,17	97,85	76,82	71,59

Kirjastolle asetettiin ohjelmistovaatimuksina suorituskyky, käytön helppous, siirrettävyys ja laajennettavuus, joihin kirjaston toteutuksessa pyrittiin vastaamaan. Kirjaston suorituskykyä esittävissä tuloksissa käsitellään ainoastaan karsintavaiheen sisältäviä testitilanteita. Suorituskykyä arvioitiin tutkimalla, miten testisovelluksen pääsilmukkaan eli yhden ruudun päivitykseen kuluva aika, jakautuu törmäysten havaitsemiseen kuluvan ajan ja muiden vaiheiden ajankäytön suhteen. Arvioinnissa käytettiin taulukoissa 5.3.–5.5. esitettyjä törmäysten havaitsemiseen kuluneita keskimääräisiä aikoja ja taulukossa 5.6. esitettyjä keskimääräisiä ruudun päivitysaikoja.

Taulukko 5.6. *Keskimääräiset ruudun päivitysajat selainkohtaisesti.*

Selain	Kappalemäärä	karsinta vmax = 20 (ms)	karsinta vmax = 5 (ms)	karsinta vmax = 1 (ms)
Firefox	10	1,5	1,4	1,4
	100	10,2	10,1	9,7
	1000	167,35	124,2	111,3
Chrome	10	0,1	0,05	0,05
	100	5,2	4,9	4,1
	1000	72,6	62,45	59,3
Opera	10	1,65	1,15	1,1
	100	10,15	10,1	10,05
	1000	137,4	115,75	111,5

Keskimääräinen ruudun päivitysaika mitattiin kuten törmäysten havaitsemiseen kulunut keskimääräinen aika. Kuvassa 5.3. on esitetty törmäysten havaitsemiseen ja muuhun toimintaan kuluneen ajan jakautuminen pääsilmukan suorituksessa.



Kuva 5.3. *Törmäysten havaitsemiseen ja muuhun toimintaan kuluneen ajan jakautuminen ruudun päivityksessä.*

Kirjaston käytön helppoutta arvioitiin subjektiivisesti tutkimalla kirjaston käyttöön tarvittavien rajapintojen selkeyttä. Rajapintojen määrittelyssä kiinnitettiin huomioita rajapintafunktioiden selkeään nimeämiseen, minkä avulla kirjaston käyttäjää pyrittiin ohjaamaan kirjaston käyttämisessä. Myös rajapintafunktioiden dokumentoinnissa pyrit-

tiin ilmaisemaan funktioiden parametrit ja paluuarvot mahdollisimman selkeästi. JavaScript-kielen heikosta tyypityksestä johtuen muuttujan tyyppiä ei ole määritelty kiinteästi, joten dokumentoinnilla pyrittiin varautumaan tilanteisiin, joissa kirjaston käyttäjälle ei ole selvää minkä tyyppistä tietoa funktion parametrit tai paluuarvot pitävät sisällään.

Kirjaston siirrettävyyttä arvioitiin kirjaston toteutuksen yksityiskohtien ja kirjaston riippuvuuksien avulla. Kirjaston toiminta perustuu koukkujen käyttöön, mikä mahdollistaa kirjaston käyttämisen eri sovelluksien osana. Koukkujen avulla kirjastoa käyttävän sovelluksen rajapinta saadaan muutettua kirjaston vaatimaan muotoon. Kirjastoa käyttävän sovelluksen tulee kuitenkin huolehtia siitä, että sovelluksen rajapinnan paluuarvot ovat kirjaston vaatimassa muodossa. Lisäksi kirjasto käyttää toteutuksessaan glMatrix-kirjastoa ja vaatii siten sen liittämistä sovellukseen.

Kirjaston laajennettavuutta arvioitiin tutkimalla kirjastoon tehtäviä muutoksia tilanteessa, jossa kirjastoon lisätään uusi rajauslaatikko. Uuden rajauslaatikon toteutus luodaan **bounding_volume**-luokasta periytettyyn aliluokkaan. Rajauslaatikolle on määriteltävä tyyppi, jota käytetään kirjaston sisäisessä toiminnassa esimerkiksi oikean leikkausalgoritmin etsimisessä. Luodun rajauslaatikon tyyppi on rekisteröitävä **bounding_volume_factory**-luokassa, jotta kirjasto pystyy luomaan olioita tästä rajauslaatikosta. Uudelle rajauslaatikolle luodaan tarvittavat leikkausalgoritmit periyttämällä **intersect_tester**-luokasta ja lopuksi uuden rajauslaatikon käyttämät leikkausalgoritmit lisätään **collision_system**-luokkaan. Uuden rajauslaatikon lisääminen vaatii siis muutoksia kahteen kirjaston luokkaan ja uusien luokkien lisäämistä. Uusia luokkia ovat rajauslaatikon määrittelyn sisältävä rajauslaatikkoluokka ja yksittäiset leikkausalgoritmit. Kirjaston toiminta mahdollistaa myös yhden leikkausalgoritmin käyttämistä kaikkien erityyppisten rajauslaatikoiden tapauksessa, mikä koettiin varteenotettavaksi ominaisuudeksi.

6 YHTEENVETO

Webin kehityksen myötä kolmiulotteinen grafiikka siirtyy työpöytäympäristöstä webiin. Kolmiulotteisen grafiikan esittäminen webissä on ollut mahdollista erillisten plugin-komponenttien avulla, mutta HTML5-standardin mukanaan tuomat uudistukset poistavat näiden plugin-komponenttien tarpeen. Kolmiulotteista grafiikkaa voidaan esittää suoraan web-sivuilla WebGL-tekniikan avulla niissä selaimissa, jotka toteuttavat standardin vaadittavilta osin.

Törmäysten havaitseminen on tärkeä osa virtuaalisia ympäristöjä, koska se mahdollistaa vuorovaikutuksen sekä ympäristön sisällä että käyttäjän ja ympäristön välillä. Yleisesti käytetty menetelmä törmäysten havaitsemiseen on käyttää yksinkertaisia geometrisia muotoja, rajauslaatikoita, ja niistä muodostettuja hierarkioita. Näin voidaan approksimoida avaruuden kappaleiden todellista geometriaa ja selvittää kappaleiden väliset törmäykset yksinkertaisten geometristen muotojen avulla. Rajauslaatikoiden avulla saavutetaan kompromissi törmäysten havaitsemisen nopeuden ja tarkkuuden välillä. Rajauslaatikoiden avulla on siis mahdollista vastata sovellusten reaaliaikaisuuden ja interaktiivisuuden tuomiin vaatimuksiin.

Kolmiulotteisen grafiikan siirtyminen webiin asettaa selainympäristöille vaatimuksia myös törmäysten havaitsemiseen liittyen. Selainten soveltuvuutta kolmiulotteisen grafiikan esittämiseen törmäysten havaitsemisen näkökulmasta tutkittiin toteuttamalla törmäysten havaitsemiseen tarkoitettu kirjasto JavaScript-kielellä. Kirjaston suorituskykyä testattiin testisovelluksella neljällä eri selaimella ja kirjaston toimintaa muiden sille asetettujen ohjelmistovaatimusten osalta arvioitiin subjektiivisesti. Kirjaston testaukseen ja arviointiin perustuen havaittiin, että toteutettu kirjasto vastaa hyvin sille asetettuihin ohjelmistovaatimuksiin ja kirjasto soveltuu hyvin törmäysten havaitsemiseen selainympäristössä.

Käytetyistä selaimista Internet Explorer ei tukenut testisovelluksessa käytettyä WebGL-tekniikkaa, joten sen osalta selaimen suorituskyvystä ei saatu mitattua tuloksia, Muiden selainten osalta törmäysten havaitsemiseen kulunut aika kasvoi kaikissa selaimissa kappalemäärän kasvaessa. Lisäksi havaittiin, että kappalemäärän kasvattaminen lisäsi myös törmäysten havaitsemiseen kulunutta suhteellista aikaa sovelluksen muuhun toimintaan nähden.

Törmäysten havaitsemiseen kuluvaa aikaa voitiin pienentää huomattavasti käyttämällä törmäysten havaitsemisessa karsintavaihetta ennen tarkkaa törmäysten havaitsemista. Karsintavaiheen vaikutusta törmäysten havaitsemisen nopeuteen voitiin muuttaa rajoittamalla kappaleiden maksiminopeutta avaruudessa. Tämä heijastuu kirjaston karsintavaiheessa käytettyyn pyyhkäise ja karsi -menetelmän toimintaan, koska se hyödyn-

tää toiminnassaan koherenssia. Mitä lyhyemmän matkan kappaleet liikkuvat avaruudessa, sitä todennäköisemmin karsintaan käytettävät listat ovat järjestyksessä tai lähes järjestyksessä. Jos kappaleiden nopeutta avaruudessa voidaan sovelluksen puolesta rajoittaa, voi törmäysten havaitsemiseen käytettävän kirjaston toiminta nopeutua.

Myös käytettävä selain vaikutti kirjaston suorituskykyyn ja selainten välillä oli selkeitä suorituskykyeroja. Google Chrome oli ympäristönä selvästi muita selaimia nopeampi suoritettujen testitilanteiden suorittamisessa. Useimmissa testitapauksissa toiseksi parhaan tuloksen sai Mozilla Firefox, jossa testin suorittamiseen kulunut aika oli 1,4–54-kertainen verrattuna Google Chrome -selaimeen. Hitaimmaksi havaitulla Operalla testin suorittamiseen kului testitilanteesta riippuen 1,6–99-kertainen aika verrattuna Google Chrome -selaimeen. Internet Explorer ei tarjonnut tukea WebGL-tekniikalle, joten mitattavia tuloksia ei saatu.

Kohdeselaimella havaittiin olevan huomattava vaikutus sovelluksen nopeuteen, vaikka käytettävä laitteisto oli identtinen. Tämä tuo haasteita sovelluskehitykseen, koska selain voi vaikuttaa joissain tapauksissa sovelluksen reaaliaikaiseen ja interaktiiviseen luonteeseen ja saattaa siten tarjota käyttäjälle tarkoitetusta poikkeavan käyttökokemuksen.

Esitettyihin tuloksiin liittyy kuitenkin joitain epävarmuustekijöitä. Työn tulokset pohjautuvat vain valittuihin ja yleisesti Windows-ympäristössä käytössä oleviin selaimiin, joten työn tulokset eivät edusta kattavasti kaikkien käytössä olevien selainten välisiä eroja. Työssä toteutetun törmäysten havaitsemiseen tarkoitetun kirjaston suorituskykymittauksessa käytettiin selainten JavaScript-ympäristöä, mikä aiheuttaa virhettä tehtyihin mittauksiin. Joissakin tapauksissa mitatuksi ajaksi saatiin nolla millisekuntia, mikä ei kerro todellista törmäysten havaitsemiseen kulunutta aikaa. Lisäksi kirjaston muita ohjelmistovaatimuksia arvioitiin ainoastaan subjektiivisesti ilman erillisiä mittareita.

Selainten välillä havaittiin selkeitä suorituskykyeroja, joten web-sovelluskehityksessä tulisi suorittaa kattavaa testausta eri selainympäristöissä. Näin voidaan varmistua, että sovellus toimii eri selaimissa tarkoituksenmukaisesti. Törmäysten havaitsemiseen kuluneen ajan osuus testisovelluksen pääsilmukan suorittamiseen kuluvasta ajasta kasvoi kappalemäärän lisääntyessä. Tästä johtuen tulevaisuudessa olisi hyödyllistä tutkia menetelmiä, joilla törmäysten havaitseminen voidaan siirtää laitteiston, kuten esimerkiksi näytönohjaimen, suoritettavaksi, jolloin selainten JavaScript-ympäristöjä voidaan käyttää vain sovelluksen peruslogiikan suorittamiseen.

LÄHTEET

Adobe. Adobe Flash Platform. URL: http://www.adobe.com/uk/flashplatform/. Viitattu: 22.8.2011.

Anttonen, M. & Salminen, A. 2011. Lively Goes 3D Report – Building 3D WebGL Applications. Technical report. Tampere University of Techonology. 42 s.

Anttonen, M., Salminen, A., Mikkonen, T. & Taivalsaari, A. 2011. Transforming the Web into a Real Application Platform: New Technologies, Emerging Trends and Missing Pieces. In Proceedings of the 26th ACM Symposium on Applied Computing, SAC'2011, TaiChung, Taiwan, March 21-25, 2011. ACM Press, proceedings Vol. 1. pp. 800–807.

Beneš, B. & Villanueva, N. 2005. GI-COLLIDE - Collision Detection with Geometry Images. In Proceedings of the 21st Spring Conference on Computer Graphics, Budmerice, Slovakia, May 12-14, 2005. NY, USA, ACM New York. pp. 95–102.

Cohen, J., Lin, M., Manocha, D. & Ponamgi, M. 1995. I-COLLIDE: An Interactive and Exact Collision Detection System for Large-Scale Environments. In Proceedings of the 1995 Symposium on Interactive 3D Graphics, Monterey, California, USA, April 9-12, 1995. NY, USA, ACM New York. pp. 189–218.

Crockford, D. 2008. JavaScript: The Good Parts. O'Reilly Media, Inc. 176 p.

Curtis, S., Tamstorf, R. & Manocha, D. 2008. Fast Collision Detection for Deformable Models Using Representative-Triangles. In Proceedings of the 2008 Symposium on Interactive 3D Graphics and Games, Redwood City, California, USA, February 15-17, 2008. NY, USA, ACM New York. pp. 61–69.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. 395 p.

GLMatrix. High performance matrix and vector operations for WebGL. URL: http://code.google.com/p/glmatrix. Viitattu: 24.8.2011.

Gottschalk, S., Lin, M. & Manocha, D. 1996. OBBTree: A Hierarchical Structure for Rapid Interference Detection. In Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques. NY, USA, ACM New York. pp. 171–180.

Haikala, I. & Märijärvi, J. 2006. Ohjelmistotuontanto. Talentum Media Oy. Jyväskylä. 440 s.

HTML. 1999. HTML 4.01 Specification. December 24, 1999. URL: http://www.w3.org/TR/html401/. Viitattu: 20.9.2011.

Hubbard, P. 1993. Interactive Collision Detection. In Proceedings of IEEE Symposium on Research Frontiers in Virtual Reality, San Jose, CA, USA, October 25-26, 1993. IEEE. pp. 24–31.

Jiménez, P., Thomas, F. & Torras, C. 2001. 3D Collision Detection: a Survey. Computers & Graphics. Vol. 25:2. pp. 269–285.

Khronos Group. 2011. WebGL Specification. Version 1.0. February 10, 2011. URL: https://www.khronos.org/registry/webgl/specs/1.0/. Viitattu: 23.5.2011.

Khronos Group. OpenGL ES 2.X and the OpenGL ES Shading Language for programmable hardware. URL: http://www.khronos.org/opengles/2_X/. Viitattu: 31.5.2011.

Koskimies, K. & Mikkonen, T. 2005. Ohjelmistoarkkitehtuurit. Talentum Media Oy. Jyväskylä. 250 s.

Liu, R., Zhang, H. & Busby, J. 2008. Convex Hull Covering of Polygonal Scenes for Accurate Collision Detection in Games. ACM International Conference Proceeding Series; Vol. 322, Windsor, Ontario, Canada, May 28-30, 2008. Toronto, Ont., Canada. Canadian Information Processing Society. pp. 203–210.

O3D. O3D Technical Overview. URL: http://code.google.com/intl/fi-FI/apis/o3d/docs/techoverview.html. Viitattu: 22.8.2011.

Paulson, L.D. 2007. Developers shift to dynamic programming languages. IEEE Computer, February 2007 Vol. 40:2. IEEE Computer Society Press. pp. 12–15.

Puhakka, A. 2008. 3D-Grafiikka. Talentum Media Oy. Helsinki. 453 s.

Quinlan, S. 1994. Efficient Distance Computation Between Non-Convex Objects. In Proceedings of IEEE International Conference on Robotics and Automation, Vol.4, San Diego, CA, USA, May 8-13, 1994. IEEE. pp. 3324–3329.

Rintala, M. & Jokinen, J. 2005. Olioiden ohjelmointi C++:lla. Talentum Media Oy. Jyväskylä. 377 s.

Schneider, P. & Eberly, D. 2003. Geometric Tools for Computer Graphics. Chapter 11. Elsevier Science. pp. 481–662.

Sylvester. Vector and Matrix math for JavaScript. URL: http://sylvester.jcoglan.com. Viitattu: 24.8.2011.

Taivalsaari, A., Mikkonen, T., Ingalls, D. & Palacz, K. 2008. Web Browser as an Application Platform: The Lively Kernel Experience. Sun Microsystems Laboratories Technical Report TR-2008-175, Jan. 2008. 22 p.

VRML97. The Information technology -- Computer graphics and image processing -- The Virtual Reality Modeling Language (VRML) -- Part 1: Functional specification and UTF-8 encoding. URL: http://www.web3d.org/x3d/specifications/vrml/ISO-IEC-14772-VRML97. Viitattu: 22.8.2011.

W3C. 2005. Document Object Model (DOM). January 19, 2005. URL: http://www.w3.org/DOM. Viitattu: 22.8.2011.

X3D. Information technology — Computer graphics, image processing and environmental representation — Extensible 3D (X3D) — Part 1: Architecture and base components. URL: http://www.web3d.org/x3d/specifications/ISO-IEC-19775-1.2-X3D-AbstractSpecification/index.html. Viitattu: 22.8.2011.